ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КІБЕРБЕЗПЕКИ ТА ЗАХИСТУ ІНФОРМАЦІЇ

КАФЕДРА СИСТЕМ ТА ТЕХНОЛОГІЙ КІБЕРБЕЗПЕКИ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Система безпечної аутентифікації користувачів з використанням JWT токенів»

зі спеціальності *125 Кібербезпека*
(код, найменування спеціальності)

освітньо-професійної програми *Інформаційна та кібернетична безпека*
(назва програми)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____Орест МАЯЦЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи БСД-42
МАЯЦЬКИЙ Орест
(прізвище, ім'я)

Керівник д. т. н., професор ГАЙДУР Галина
(науковий ступінь, вчене звання, прізвище, ім'я)

Рецензент _____
(науковий ступінь, вчене звання, прізвище, ім'я)

Київ 2025

ВСТУП

Актуальність дослідження. Сучасні інформаційні системи стикаються зі зростаючою кількістю кіберзагроз, серед яких особливе місце займають атаки на системи аутентифікації. Надійний механізм перевірки автентичності користувачів є критично важливим елементом захисту будь-якої IT-інфраструктури. У сучасних умовах глобального зростання екосистеми API, традиційні методи аутентифікації на основі сесій стають менш ефективними. Це зумовлено необхідністю забезпечення масштабованості систем, зростанням популярності мікросервісних архітектур, вимогами до кросплатформної сумісності, а також активним використанням хмарних сервісів та мобільних додатків, де класичні механізми управління сесіями можуть створювати значні обмеження. Крім того, постійне підвищення вимог до безпеки з боку нормативно-правових актів та міжнародних стандартів, таких як GDPR та NIST, вимагає впровадження більш надійних та стійких до атак механізмів аутентифікації.

Саме в цьому контексті Json Web Token (JWT) виступають сучасним рішенням для забезпечення безпечної аутентифікації. Їх переваги включають відсутність необхідності зберігати стан сервера, можливість передачі додаткової інформації про користувача, простоту інтеграції з різними платформами та мовами програмування. Крім того, JWT ефективно використовуються у Zero Trust-архітектурах, які дедалі частіше впроваджуються у корпоративних середовищах для мінімізації ризиків несанкціонованого доступу. Особливу актуальність дослідження набуває у зв’язку з поширеними помилками впровадження JWT, які можуть призвести до компрометації токенів, несанкціонованого доступу до системи, втрати конфіденційності даних, а також до можливості реалізації атак, таких як повторне використання токенів (replay attack) або підробка цифрового підпису (signature spoofing).

Вищесказане визначає актуальність теми даної кваліфікаційної роботи, основний зміст якої становлять дослідження методів та засобів безпечної аутентифікації користувачів з використанням JWT токенів.

Об'єкт дослідження – аутентифікація користувачів.

Предмет дослідження – методи та засоби створення безпечної системи аутентифікації користувачів з використанням JWT токенів.

Мета роботи: розробка практичних рекомендацій щодо створення та використання системи безпечної аутентифікації користувачів з використанням JWT токенів

Наукові завдання:

дослідити сутність проблеми безпечної автентифікації в інформаційних системах;

проаналізувати основні підходи до автентифікації та управління доступом;

проаналізувати існуючі рішення з автентифікації на основі JWT-токенів;

проаналізувати методи та засоби захисту JWT-токенів від компрометації та атак;

запропонувати варіант системи безпечної автентифікації з використанням JWT-токенів;

розробити рекомендації щодо застосування методів і засобів безпечної автентифікації за допомогою JWT-токенів.

Методи дослідження – Аналіз наукових джерел, стандартів та літератури за даною темою, документації щодо JWT, та механізмів автентифікації. Оцінка JWT відносно інших методів аутентифікації. Реалізація JWT-аутентифікації у Flask-додатку з аналізом продуктивності та безпеки.

Практичне значення одержаних результатів: розроблено рекомендації щодо застосування методів та засобів безпечної автентифікації за допомогою JWT-токенів, а також рекомендації фахівцям з кібербезпеки щодо їх реалізації в інформаційних системах.

Результати кваліфікаційної роботи апробовані на Всеукраїнській науково-практичній конференції «Цифрова трансформація кібербезпеки», яка відбулася 24 квітня 2025 року в Державному університеті інформаційно-комунікаційних технологій, м. Київ.

1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ БЕЗПЕЧНОЇ АВТЕНТИФІКАЦІЇ

1.1 Аналіз загроз та вразливостей у сучасних системах автентифікації

Сьогодні Інтернет став невід'ємною частиною повсякденного життя, проникаючи в усі його сфери – від спілкування в соціальних мережах до фінансових операцій. Величезні обсяги важливої інформації, втрата якої може призвести до серйозних наслідків, потребують надійного захисту. Однак постає проблема балансу між механізмами безпеки та зручністю їх використання. Надмірно складні системи захисту іноді ускладнюють доступ навіть авторизованим користувачам, що знижує ефективність роботи. Сучасні методи захисту даних спрямовані на забезпечення конфіденційності, цілісності та доступності інформації, попри різноманітні загрози. Вони дозволяють мінімізувати ризики витоку або крадіжки критично важливих даних. Враховуючи це, можна стверджувати, що інформаційна безпека в онлайн-середовищі є однією з найважливіших складових сучасних технологій.

Автентифікація – це процес перевірки достовірності суб'єкта, який намагається отримати доступ до системи. Вона тісно пов'язана з ідентифікацією (встановленням унікальності користувача) та авторизацією (наданням дозволу на виконання певних дій після підтвердження особи).

Основною метою автентифікації є забезпечення того, щоб доступ до ресурсу отримували лише ті суб'єкти, які мають на це відповідні повноваження.

Розвиток цифрових технологій супроводжується збільшенням кількості та складності атак на системи автентифікації:

- фішинг (Phishing) – один із найпоширеніших методів соціальної інженерії, що полягає у введенні користувача в оману з метою отримання його облікових даних. Наприклад, користувачу надсилають фальшиве повідомлення, яке імітує

офіційний лист від банку або іншої організації, з проханням ввести логін і пароль на підробленому сайті;

- брутфорс (Brute-force) та словникові атаки – атаки базуються на масовому переборі паролів або використанні списків поширених комбінацій. Вразливими до таких атак є системи, які не застосовують обмеження кількості спроб входу або не використовують CAPTCHA;
- витік облікових даних унаслідок компрометації сторонніх сервісів (наприклад, у результаті SQL-ін'єкцій чи злому бази даних), облікові записи потрапляють до відкритих джерел, після чого зловмисники можуть використати їх для атак на інші ресурси (атака повторного використання паролів – password reuse);
- захоплення сесії (Session Hijacking) – викрадення ідентифікатора сесії користувача (наприклад, через Cross Site Scripting, XSS), що дозволяє зловмиснику діяти від його імені без знання логіна і пароля;
- атаки типу людина посередині (MitM) – якщо автентифікаційний трафік передається без шифрування або з використанням застарілих протоколів (наприклад, HTTP замість HTTPS), його може бути перехоплено та змінено в реальному часі.

Навіть за наявності надійного пароля система може бути вразливою, якщо неправильно організовано його зберігання:

- зберігання паролів у відкритому вигляді (plaintext) є критичною помилкою;
- недостатнє хешування (наприклад, використання швидких алгоритмів MD5 або SHA1 без солі)[1];
- відсутність захисту від витоку бази паролів (відсутність шифрування сховищ, недбале адміністрування).

Сучасні системи автентифікації, зокрема ті, що побудовані на принципах REST API, часто використовують JWT-токени. Вони також мають свої специфічні загрози[2]:

- крадіжка токена – якщо токен зберігається у localStorage, його можуть викрасти через XSS-уразливість;

- replay-атаки – якщо термін дії токена надто довгий, зловмисник може використати його повторно;
- використання слабкого або неправильного алгоритму підпису (наприклад, «none»);
- збереження секретного ключа в клієнтському коді або витік ключа з серверної частини.

Окрім загроз, пов’язаних із паролями та токенами, існує низка інших вразливостей, які можуть бути використані для компрометації системи автентифікації, серед яких: атака через підробку міжсайтових запитів (CSRF) - атака, яка змушує браузер користувача виконати небажаний запит до сервера, на якому користувач автентифікований. Це можливо, якщо сесія або токен передаються через Cookie.

Деякі вразливості виникають не через зовнішні фактори, а внаслідок неправильного проєктування або реалізації самої логіки автентифікації:

- відсутність обмеження спроб входу – відкриває можливості для брутфорс-атак;
- невірна обробка токенів – відсутність перевірки підпису або дати закінчення дії (exp) у JWT;
- фіксовані секрети – застосування одного статичного секретного ключа для всіх токенів є небезпечною практикою;
- відсутність механізму відкликання токенів – у системах із JWT часто складно реалізувати відкликання токенів, особливо access-токенів (токенів виданих для ідентифікації), без використання чорного списку (blacklist) або зберігання стану на сервері.

Open Web Application Security Project (OWASP) – міжнародна організація, яка публікує щорічний рейтинг OWASP Top 10, до якого входять найпоширеніші вразливості вебдодатків. До більш релевантних для автентифікації належать (рисунок 1.1) [3]:

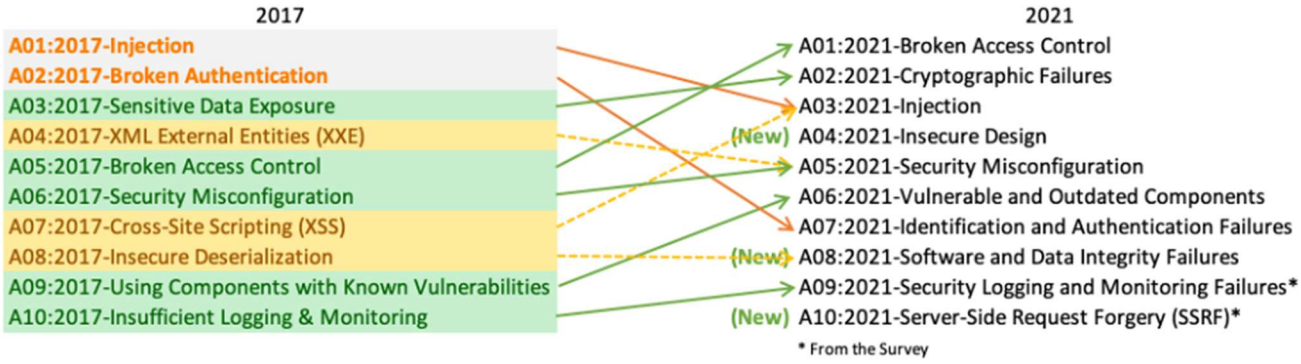

Рис. 1.1. Список найпоширеніших вразливостей вебдодатків [3]

- A01: Broken Access Control – порушення контролю доступу;
- A02: Cryptographic Failures – помилки у шифруванні, включно з неправильним зберіганням паролів;
- A07: Identification and Authentication Failures – безпосередньо помилки автентифікації.

Цей рейтинг є стандартом де-факто для безпечної розробки і повинен враховуватися під час проєктування будь-якої системи автентифікації.

Аналіз загроз і вразливостей у системах автентифікації показує, що сучасні атаки мають як технічну, так і соціальну природу. Зловмисники постійно вдосконалюють інструменти і методи, комбінуючи класичні атаки (фішинг, брутфорс, XSS, MitM) з маніпуляціями на рівні користувача (соціальна інженерія). Уразливими можуть бути не тільки кінцеві користувачі, але й самі системи, якщо:

- використовується застаріле або ненадійне шифрування;
- відсутній захист від повторного використання сесій;
- немає обмеження на кількість спроб входу;
- токени автентифікації обробляються або зберігаються неналежним чином;
- не передбачена централізована система відкликання доступу.

Таким чином, для досягнення високого рівня безпеки автентифікації необхідне комплексне поєднання:

- надійної криптографії (з урахуванням сучасних стандартів);
- коректної реалізації протоколів обміну даними;
- психологічної підготовки користувачів (підвищення обізнаності);

- використання сучасних методів захисту, зокрема багатофакторної автентифікації (MFA) та токенізації.

Саме з таких міркувань системи, засновані на використанні JWT токенів, стали особливо популярними в контексті веборієнтованої автентифікації – вони дають змогу створювати масштабовані, безстатеві (stateless) системи доступу.

Однак навіть JWT не є панацеєю – їх використання передбачає дотримання чітких принципів безпеки, таких як правильне зберігання, шифрування, управління терміном дії та механізми відкликання.

1.2 Підходи до забезпечення безпечної автентифікації

Забезпечення надійної автентифікації є основним елементом кіберзахисту в будь-якій інформаційній системі. Через неправильну реалізацію автентифікації зловмисники можуть скористатися вразливостями автентифікації, реалізувати атаки на вебзастосунок для отримання несанкційного доступу.

Усі методи автентифікації умовно можна поділити на три основні категорії:

- що знає користувач (знання) – найстаріший та найпоширеніший метод, до якого належать паролі, PIN-коди, відповіді на контрольні запитання тощо. Цей метод є простим у реалізації, проте має суттєві недоліки – люди схильні використовувати слабкі паролі, повторювати їх у різних системах, зберігати їх у незахищених місцях. Існує низка досліджень (наприклад, дослідження Google у 2019 році), які підтверджують, що понад 52% користувачів використовують один і той самий пароль на кількох платформах. Це суттєво спрощує завдання зловмисникам у разі компрометації одного з сервісів;
- що має користувач (володіння) – підхід базується на використанні об’єктів, якими володіє користувач – наприклад, смарт-карток, апаратних ключів (YubiKey, RSA SecurID), токенів, а також мобільних пристроїв із додатками для генерації OTP (наприклад, Google Authenticator або Microsoft Authenticator). Основна перевага

цього підходу – ускладнення несанкціонованого доступу, оскільки навіть у разі компрометації пароля зловмисник не матиме фізичного доступу до пристрою автентифікації;

- хто є користувач (біометрія) – підхід ґрунтується на біометричних характеристиках користувача: відбитках пальців, скануванні обличчя, райдужки ока, голосу тощо. Біометричні системи зручні у використанні та важко підроблювані, однак їхня реалізація потребує складного обладнання, а також створює ризики конфіденційності: у разі компрометації біометричних даних неможливо «змінити» відбиток пальця чи структуру обличчя.

Використання лише одного фактора (особливо «що знає») в сучасних умовах є недостатньо безпечним. Звідси виникає потреба в багатофакторній автентифікації (MFA), яка поєднує щонайменше два з цих підходів.

Одноразовий пароль (OTP) – це динамічно згенерований код, що діє короткий час або використовується лише один раз. Він може бути реалізований у вигляді:

- Time-based OTP (TOTP) – залежить від часу (Google Authenticator, FreeOTP);
- HMAC-based OTP (HOTP) – залежить від лічильника (використовується рідше).

OTP суттєво знижує ризики фішингу та компрометації паролів. MFA передбачає, що для успішного входу необхідно підтвердити щонайменше два незалежні фактори автентифікації, наприклад:

- пароль + SMS-код;
- пароль + біометричне підтвердження;
- TOTP + апаратний ключ (наприклад, YubiKey).

Важливо зазначити, що не всі типи MFA є однаково безпечними. Наприклад, SMS як другий фактор може бути перехоплено внаслідок атаки типу SIM swap. Більш безпечним варіантом є апаратні токени або додатки TOTP.

Незалежно від конкретного методу, побудова безпечної системи автентифікації повинна відповідати таким принципам:

- надійність ідентифікації – система повинна точно встановлювати особу користувача;
- уникнення зберігання чутливих даних у відкритому вигляді – паролі повинні зберігатись у вигляді хешів із використанням криптостійких алгоритмів (наприклад, bcrypt, Argon2);
- обмеження на кількість спроб входу – запобігає атакам перебору (brute force);
- протидія автоматизованим спробам – використання CAPTCHA, таймерів, систем виявлення ботів;
- контроль геолокації та пристрою – нестандартна поведінка користувача може сигналізувати про злом;
- прозорість і аудит – усі події автентифікації повинні логуватися.

Підходи до автентифікації зазнали значної еволюції: від простих паролів до складних багатофакторних і адаптивних схем із використанням токенів та біометрії. Сучасні протоколи, такі як OAuth 2.0, OpenID Connect, SAML, а також криптографічні методи, дозволяють створювати надійні, масштабовані та гнучкі рішення автентифікації, адаптовані до загроз XXI століття. Проте навіть найкращі технології не є панацеєю без правильної реалізації, постійного аудиту, моніторингу та навчання користувачів. Комбінація технічних, організаційних і поведінкових підходів є ключем до створення по-справжньому безпечного середовища.

1.3 Аналіз методів автентифікації та їхня ефективність

Процес автентифікації користувача починається з етапу ідентифікації. На цьому етапі система отримує унікальний ідентифікатор, який представляє конкретного користувача в системі. Цей ідентифікатор (наприклад, логін або ID акаунта) дозволяє системі знайти відповідний запис у своїй базі даних. Однак сам факт наявності такого ідентифікатора ще не гарантує, що особа, яка його використовує, є тим, за кого себе видає. Для підтвердження справжності

користувача необхідний процес автентифікації, під час якого пред’являються додаткові докази (облікові дані). Найпоширенішим прикладом таких даних є логін та пароль. Система порівнює надані дані зі збереженими в своїй базі і лише у разі їх відповідності підтверджує автентичність користувача. Успішна автентифікація відкриває доступ до системи згідно з призначеними для даного користувача правами. Цей механізм є ключовим елементом безпеки будь-якого сучасного вебзастосунку, оскільки саме він забезпечує захист від несанкціонованого доступу до конфіденційної інформації та функціоналу системи.

Найбільш використовуваними є такі методи автентифікації [1]:

- автентифікація на основі файлів cookie;
- втентифікація за допомогою токенів;
- сторонній доступ (OAuth, токен API);
- OpenID;
- SAML.

HTTP Cookie session – це важливий крок у розвитку веб-технологій, який дозволяє подолати основне обмеження HTTP – відсутність збереження стану (stateless). Оскільки HTTP-запити не зберігають інформацію про попередні взаємодії між клієнтом і сервером, виникла потреба у механізмі, який би дозволяв ідентифікувати користувачів і зберігати дані їхніх сеансів. Саме для цього були розроблені HTTP-сесії, що дають змогу серверу зберігати тимчасові дані про користувача під час його роботи з веб-додатком.

Основна ідея сесій полягає в тому, що сервер створює унікальний ідентифікатор для кожного користувача (Session ID) і передає його клієнту, зазвичай через cookie. Після цього при кожному наступному запиті браузер автоматично відправляє цей ідентифікатор назад на сервер, що дозволяє відновити дані сесії та продовжити роботу з користувачем без необхідності повторної автентифікації.

Сесії можуть містити різноманітну інформацію, таку як стан входу в систему, налаштування користувача, тимчасові дані форм або інші параметри, необхідні для роботи веб-додатка. Однак, незважаючи на зручність, використання cookie-сесій

має свої обмеження. Наприклад, cookie займають місце в браузері, а при великій кількості активних користувачів серверу доводиться зберігати значні обсяги даних сесій, що може призводити до збільшення навантаження.

Крім того, cookie можуть бути вразливі до атак, таких як викрадення сесії (session hijacking) або міжсайтові запити (CSRF), тому для забезпечення безпеки важливо використовувати додаткові заходи, такі як HTTPS, HttpOnly та Secure-прапорці для Cookie, а також регулярне оновлення ідентифікаторів сесій.

Незважаючи на це, HTTP Cookie session залишаються одним із найпоширеніших методів керування сеансами в сучасних веб-додатках, оскільки вони прості у реалізації, добре підтримуються браузерами та дозволяють створювати персоналізований взаємодію з користувачами. У майбутньому їх можуть доповнювати або замінювати більш сучасні технології, такі як JWT або OAuth, але наразі вони продовжують грати ключову роль у веб-розробці. Далі наведено переваги та недоліки автентифікації на основі HTTP Cookie session у таблиці 1.1.

Таблиця 1.1.

Переваги та недоліки автентифікації на основі HTTP Cookie session

Переваги	Недоліки
Файли Cookie займають мінімум місця.	Файли Cookie можуть бути ціллю для XSS (міжсайтовий скриптинг) та CSRF (міжсайтова підробка запитів).
Файли Cookie легко реалізуються та керуються.	при великій кількості користувачів можуть виникати складнощі з обробкою Cookie.
Дані зберігаються на сервері, що ускладнює їх викрадення або підміну.	Часто файли Cookie містять конфіденційну інформацію, що робить їх привабливими для зловмисників.
Інформацію в Cookie зазвичай шифрують перед передачею, а самі дані передаються через захищений протокол HTTPS.	Безпека Cookie залежить від розробників, які можуть допустити помилки в конфігурації.

На відміну від традиційних сесійних cookie, автентифікація за допомогою токенів не має уніфікованого шаблону реалізації, що призводить до створення

індивідуальних рішень для кожної системи. Цей метод особливо поширений у розподілених системах єдиного входу (SSO), де провайдер послуг (service provider) передає функцію перевірки особистості спеціалізованому сервісу (identity provider) [1].

Типовий приклад – авторизація через соціальні мережі, де платформи (як Facebook чи Google) виступають провайдерами ідентифікації, а клієнтський додаток приймає їхні верифікаційні токени. Механізм роботи передбачає генерацію identity provider'ом цифрового сертифіката (токена), який містить достовірні дані користувача. Цей токен потім використовується service provider'ом для всіх етапів роботи з користувачем – від ідентифікації до визначення рівнів доступу.

Сучасні стандарти (наприклад, OAuth 2.0) формалізують цей процес делегованої автентифікації. Однак існують і альтернативні схеми – зокрема, реалізації з JWT-токенами, де сервер самостійно генерує та валідує токени без участі зовнішніх identity provider'ів. У таких випадках серверна частина виконує повний цикл роботи з токенами – від створення до перевірки отриманих від клієнта даних.

JWT є сучасним механізмом автентифікації, який використовує структуровані дані у форматі JSON. Головною особливістю цієї технології є наявність цифрового підпису, що дозволяє перевіряти справжність переданої інформації. На відміну від традиційних методів, JWT дозволяє передавати значно більший обсяг даних у порівнянні зі звичайними cookie-файлами [4]. Коли сервер отримує JWT, він може бути впевненим у достовірності даних, оскільки токен містить криптографічний підпис, що унеможливлює зміну інформації посередниками після відправлення. JWT гарантує цілісність даних, але не їх шифрування – вміст JSON у токені залишається відкритим і може бути переглянутий у разі перехоплення. Тому настійно рекомендується використовувати JWT тільки через захищене з'єднання HTTPS. Однією з найскладніших задач при реалізації JWT є вибір місця зберігання токена, і рівень безпеки автентифікації залежить від правильних налаштувань розробника. Токен

потрібно зберігати у безпечному місці в браузері користувача: localStorage або sessionStorage, Cookie-файлі чи в пам’яті застосунку. Якщо токен зберігається в localStorage або sessionStorage, він доступний будь-якому скрипту на сторінці, що робить його вразливим до атак типу Cross-site scripting (XSS). Тому зберігання токена в цих сховищах не рекомендується. При зберіганні токена в Cookie-файлах зростає ризик атак Cross-site request forgery (CSRF), через що також не бажано їх використовувати. Якщо будь-який скрипт на сторінці буде скомпрометований, зловмисник зможе отримати доступ до всіх токенів, що зберігаються в браузері. Найбезпечнішим варіантом є зберігання токена безпосередньо в пам’яті застосунку, до якої неможливо дістатися стандартними методами, або використання HttpOnly Cookie – цього виду Cookie не загрожують CSRF-атаки. Далі наведені переваги та недоліки використання JWT-токена у таблиці 1.2.

Таблиця 1.2.

Переваги та недоліки використання JWT-токена

Переваги	Недоліки
Збільшення кількості користувачів не впливає на масштабованість сервера.	JWT має значно більший розмір порівняно з ідентифікатором сеансу у Cookie.
Відсутність стану, сесії не зберігаються на сервері, що знижує серверне навантаження, підвищує продуктивність і забезпечує кращу масштабованість.	JWT не дозволяє безпосередньо відкликати доступ користувача після видачі токена.
Відсутність стану забезпечує можливість автентифікації та авторизації між різними доменами та платформами.	Токен зберігається на клієнті, що робить його вразливим для зловмисників.
Адаптованість передбачає наявність у токені будь-якої інформації про вебзастосунок, що дозволяє здійснювати точний і персоналізований контроль доступу.	Якщо зловмисник отримає JWT користувача або системи, він зможе використати його для викрадення особистості та отримання доступу до їхніх ресурсів.

Для підвищення безпеки автентифікації на основі JWT-токена був розроблений принцип розмежування між видами токенів на Access і Refresh [3]. Такий підхід дозволяє досягти балансу між зручністю користування та

контрольованою безпекою. Access-токен використовується для авторизації запитів до захищених ресурсів і має короткий строк дії (наприклад, 15 хвилин). У стандартній ситуації Access-токени не зберігаються на сервері (оскільки є stateless), однак у випадках примусового виходу користувача (logout до завершення терміну дії) або при блокуванні токенів адміністративно, їх слід заносити до тимчасового сховища (наприклад, Redis), щоб унеможливити подальше використання. Refresh-токен має довший термін дії (наприклад, 1 день) і застосовується для отримання нової пари Access/Refresh токенів. Та само, як Access-токен, Refresh-токен може бути збережений у тимчасовому сховищі для підтримки механізмів відкликання (logout, revoke) або контролю повторного використання (replay attack).

Браузерний fingerprint – це потужний механізм ідентифікації користувачів у цифровому середовищі, який ґрунтується на унікальній комбінації технічних параметрів браузера та пристрою. На відміну від традиційних методів, таких як cookies або IP-адреси, fingerprint не вимагає зберігання даних на стороні клієнта і не може бути легко скинутий або підроблений.

Основним принципом роботи браузерного fingerprint є збір та аналіз десятків характеристик, серед яких: версія операційної системи, роздільна здатність екрана, список встановлених шрифтів, налаштування мови, часовий пояс, параметри WebGL та Canvas, апаратні особливості процесора та багато іншого. Ці дані об’єднуються в унікальний цифровий відбиток, який з високою точністю дозволяє розрізняти окремих користувачів навіть при використанні анонімізаторів або VPN. Це робить його ідеальним інструментом для таких завдань, як боротьба з шахрайством, виявлення ботів, захист від DDoS-атак (Distributed Denial of Service, DDoS) та підвищення безпеки фінансових операцій.

Однак fingerprint має і свої недоліки – зокрема, він може викликати питання щодо конфіденційності, оскільки дозволяє відстежувати користувачів без їх явної згоди. Крім того, деякі браузери (наприклад, Tor або Firefox зі спеціальними налаштуваннями) активно протидіють збору fingerprint, що може ускладнити його використання. Незважаючи на це, браузерний fingerprint залишається одним із найнадійніших способів ідентифікації в інтернеті, особливо в комбінації з іншими

методами, такими як двофакторна автентифікація або поведінкові біометричні дані. У майбутньому, з розвитком технологій машинного навчання та аналізу даних, його точність і сфера застосування лише зростатимуть.

SAML – це стандарт автентифікації та авторизації, варіант розширюваної мови розмітки (Extensible Markup Language, XML) для обміну інформацією про безпеку в інтернеті. У протоколі SAML обмін інформацією здійснюється між двома ключовими учасниками: стороною, що засвідчує (Identity Provider, IdP), та стороною, що довіряє отриманим даним (Service Provider, SP) [1, 3]. Ідентифікаційні дані передаються у вигляді підтверджень безпеки – структурованих XML-документів, які містять рішення щодо доступу користувача до ресурсів. У рамках протоколу реалізовано два основні підходи до єдиного входу (Single Sign-On, SSO): з ініціативи IdP та з ініціативи SP. В обох випадках саме IdP виконує автентифікацію користувача, проте відрізняється початкова точка входу. У сценарії, коли ініціатором виступає IdP, користувач спочатку автентифікується на IdP, і після цього відбувається перенаправлення до SP з відповіддю у форматі SAML. У випадку ініціативи з боку SP, користувач спочатку звертається до SP, який формує SAML-запит та пересилає його до IdP. Після успішної автентифікації IdP надсилає відповідь назад до SP, що дозволяє завершити процес входу.

OpenID Connect є протоколом автентифікації, який базується на стандарті OAuth 2.0. Його мета – забезпечити уніфікований спосіб автентифікації користувачів та передачі інформації про їхню особу до сторонніх застосунків і сервісів [1, 3]. Ключова відмінність між OAuth 2.0 та OpenID Connect полягає в характері токенів, що видаються під час взаємодії. Якщо OAuth 2.0 видає access-токени з чітко визначеним терміном дії та набором дозволів, то OpenID Connect додатково формує ідентифікаційні токени (ID tokens), що слугують для підтвердження особи користувача. Незважаючи на схожість в архітектурі та логіці роботи обох протоколів, саме розподіл функцій між токенами доступу (для авторизації) та токенами ідентифікації (для автентифікації) є принциповим і визначає відмінність у їхньому використанні.

Для автентифікації вебзастосунків використовуються кілька методів, зокрема файли Cookie, JWT, OAuth, API Token, SAML та OpenID, описані вище. Залежно від сценарію застосування та вимог безпеки вони мають різні переваги та недоліки, наведені у таблиці 1.3.

Таблиця 1.3.

Переваги та недоліки найпоширеніших методів автентифікації

Метод	Переваги	Недоліки
Cookies	Проста реалізація та інтеграція, є можливість зберігати інформацію про сеанс, налаштування або інші дані для ідентифікації користувача.	Вразливості до CSRF-атак, збільшення витрат на кожен запит, обмеження на розмір корисного навантаження та частоту запитів.
JWT	Безстатевий, містить більше даних, може підтримувати кілька доменів або служб, перевірка можлива за наявності секретного ключа.	Схильний до XSS-атак, з обмеженим терміном дії, і важко скасувати (потрібен blacklist) або оновлений (потрібен refresh-токен).
OAuth	Дозволяє делегувати доступ до ресурсів користувача між веб-сайтами, уникаючи необхідності передачі облікових даних. Може використовувати різні типи токенів.	Складна реалізація, вимагає участі багатьох сервісів, створює певні ризики для безпеки: фішингові атаки, витік токенів або їх повторне відтворення.
SAML	Дозволяє безпечно переходити між веб-сайтами без необхідності повторної автентифікації. Використовує зашифровані та підписані твердження, які містять перевірену інформацію про користувача, що дозволяє іншим сайтам довіряти його ідентичності.	Продуктивність системи обмежена через використання XML, великий розмір даних і значну кількість повідомлень. Обробка та розбір XML вимагають значних ресурсів.
OpenID	Дозволяє користувачу входити на одному ресурсі та використовувати цю аутентифікацію для доступу до інших ресурсів. Усуває необхідність створення окремих облікових записів та повторного введення облікових даних. Може використовувати різні типи токенів.	Складна реалізація, вимагає участі багатьох сервісів, створює певні ризики для безпеки: фішингові атаки, витік токенів або їх повторне відтворення.

На цей час найбільш поширеними методами автентифікації є Cookie sessions та JWT token [3]. Оскільки із загального порівняння складно зробити висновок, то більш детально порівняємо методи автентифікації Cookie та JWT. Далі наведено результати порівняння цих двох методів за такими параметрами: масштабованість, безпека, сервіси API RESTful, продуктивність (Таблиця 1.4).

Таблиця 1.4.

Параметри використання методів автентифікації Cookie та JWT token

Параметри	Cookie sessions	JWT token
Масштабованість	Інформація про сеанс користувача може зберігатися як на самому сервері, так і в базі даних. Для систем, які розширюються шляхом додавання нових серверів, використовується єдине місце для зберігання даних сеансів, до якого мають доступ всі сервери.	Легко масштабується, оскільки токен безстатевий, що економить витрати, оскільки не потрібен виділений сервер для зберігання сесій споживачів.
Безпека	Використання сесійних кукі передбачає зберігання даних на сервері, що робить їх відносно захищеними від несанкціонованого доступу.	Токен захищений підписом, який генерується на сервері за допомогою секретного ключа. Без ключа неможливо підробити або змінити токен. Зберігати токен треба у надійному сховищі.
Сервіси API RESTful	Застосунок потребує механізму збереження стану, враховуючи, що він працює в розподіленій архітектурі, де API обслуговується окремим сервером.	RESTful API – безстатеве, тож не важливо, звідки API або застосунок обслуговується.
Продуктивність	Оскільки JWT значно більший за розміром, ніж ідентифікатор SESSION, додаткові витрати на його кодування є відносно невеликими.	JWT додають overhead до кожного HTTP-запиту. Проте, цей недолік компенсується можливістю зберігати корисне навантаження (payload) на стороні клієнта, наприклад, ролі користувача або його ідентифікатор.

На основі проведеного аналізу можна дійти висновку, що механізм автентифікації за допомогою Cookie-сесій поступово втрачає актуальність у порівнянні з JWT-підходом. Для невеликих застосунків цілком достатньо базової JWT-автентифікації без використання Refresh-токена – вона здатна повністю замінити традиційні сесії, що базуються на Cookies. У випадках, коли йдеться про програми середнього або великого масштабу, доцільно реалізовувати модель з Access/Refresh-токенами, доповнюючи її перевіркою fingerprint браузера для підвищення рівня безпеки.

2 АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ АВТЕНТИФІКАЦІЇ НА БАЗІ JWT

2.1 Принципи роботи JWT токенів

JWT – це відкритий стандарт RFC 7519 для компактного і самодостатнього представлення інформації у вигляді JSON-об’єкта, який передається між сторонами з гарантією цілісності через цифровий підпис [5]. JWT-токен складається з трьох частин: заголовка (header), навантаження (payload, набір «даних» – claims) та підпису (signature), розділених крапками; кожна частина кодується у форматі URL-безпечного Base64 (base64url) [3, 6]. Завдяки цій структурі JWT є самодостатнім: він містить усю необхідну інформацію про автентифікацію або авторизацію (заявлені дані) разом із підписом, що захищає дані від несанкціонованих змін. Основні переваги JWT включають високу безпеку завдяки цифровому підпису, зручність використання в розподілених системах та відсутність необхідності зберігати стан сесії на сервері. Однак є й недоліки, такі як неможливість швидкого відкликання токена у разі компрометації та дещо збільшений розмір порівняно з іншими методами. JWT широко використовується в сучасних веб-додатках, особливо в комбінації з іншими технологіями безпеки, та є важливим інструментом для забезпечення надійної автентифікації та авторизації.

JWT складається з трьох основних компонентів (рисунок 2.1) [5, 6]:

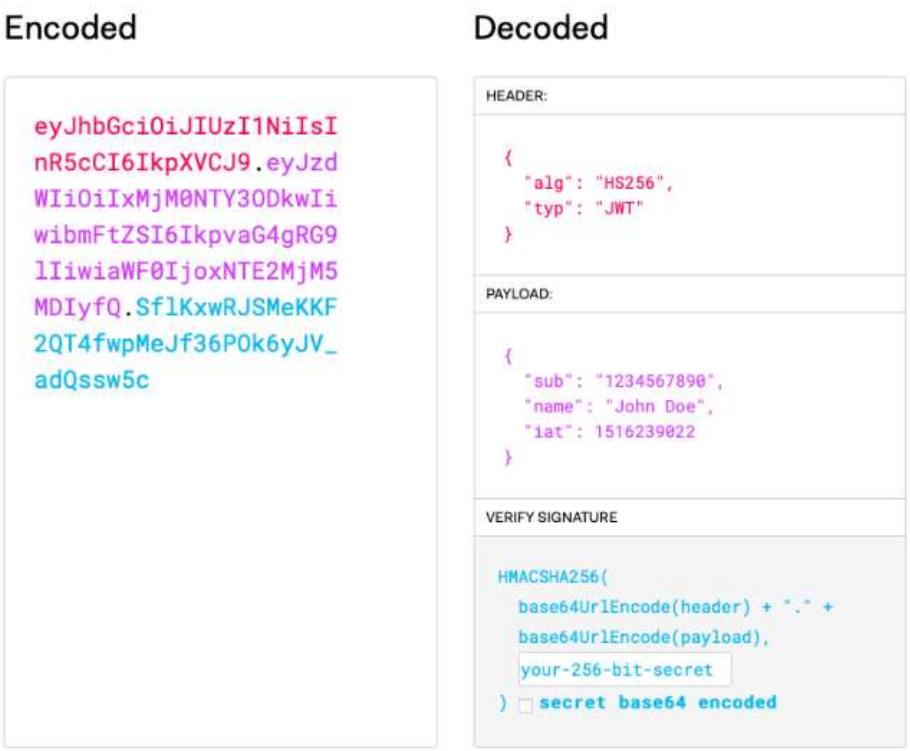

Рис. 2.1. Складові компоненти JWT [6]

- заголовок (header), JSON-об’єкт з метаданими токена. Зазвичай містить поля typ (тип токена, наприклад, JWT) і alg (алгоритм підпису, наприклад, HS256 або RS256). Наприклад, заголовок може виглядати наступним чином: {"alg":"HS256","typ":"JWT"}. Після формування JSON він кодується у Base64Url і стає першою частиною токена.
- навантаження (payload), містить набір заявлених даних (claims) у форматі JSON. Claims – це данні про суб’єкт (зазвичай користувача) та його атрибути. Існують дві категорії claims [7]:
  - зареєстровані данні (registered claims) – стандартні поля з усталеним значенням імен (Internet Assigned Numbers Authority (IANA)), які не є обов’язковими, але рекомендуються для забезпечення сумісності систем. Специфікація JWT надає сім registered claims, які не є обов`язковими але рекомендованими, серед них найпоширеніші: iss (issuer – видавець токена), sub (subject – суб’єкт токена, часто це ідентифікатор користувача), aud (audience – аудиторія, для якої призначено токен), exp (expiration – час закінчення дії токена), nbf (not before – час початку дії), iat (issued at – час видачі, може використовуватися

для обчислення часу життя токену), jti (JWT ID – унікальний ідентифікатор токена, може використовуватися для протидії повторного використання токена);

- публічні данні (public claims) – складаються з незареєстрованих публічних або приватних полів. Публічні поля стійкі до колізій, тоді як приватні піддаються можливим колізіям. Можливо визначити власні поля, і додати їх до токена. Наприклад можливо додати електронну адресу до access-токену, та використовувати це для унікальної ідентифікації користувача, або додати спеціальну інформацію, що зберігається в профілі користувача Auth0. Поки кастомні поля не будуть видалені системою на стороні серверу, при використанні токена оновлення (refresh token), новий access-токен буде с такими ж кастомними полями як і старий;

- підпис (signature) – криптографічний підпис, що обчислюється від поєднання закодованих header і payload разом з секретом (ключем) за алгоритмом, зазначеним у заголовку. Наприклад, для алгоритму HS256 підпис створюється за формулою [6]:

$$HMACSHA256(\ base64UrlEncode(header) + "." + base64UrlEncode(payload),\ secret) \quad (2.1)$$

Де, $header$ – JSON-об’єкт з метаданими токена;

$payload$ – набір заявлених claims у форматі JSON;

$secret$ – секретний ключ обох сторін.

У випадку RS256 використовується пара ключів (приватний для підпису та публічний для перевірки). Обчислений підпис кодується у Base64Url і стає третьою частиною токена. Підпис гарантує цілісність даних і автентичність JWT: при отриманні токена сервер може перевірити, що вміст (header+payload) не змінювався і що підпис справді згенеровано з наявним секретом чи приватним ключем.

Наприклад, згідно з RFC 7519, утворений токен матиме три частини, розділені крапками, де остання частина є підписом (рисунок 2.2) [6].

Рис. 2.2. Вигляд JWT токена, згідно з RFC 7519 [6]

Для підпису JWT використовуються різні криптографічні алгоритми. Найбільш поширені з них – HS256 та RS256 [7]:

- HMAC-SHA256 (HS256) – симетричний алгоритм: обидві сторони (той, хто видає токен, і той, хто його перевіряє) мають один спільний секретний ключ. Застосовуючи HMAC-SHA256, сервер формує підпис, а клієнт або довірений сервіс потім використовує той же секрет для перевірки підпису. Головний недолік HS256 полягає у потребі надійно зберігати секрет у кількох місцях – у разі його компрометації зловмисник може створювати виглядно дійсні токени. HS256 зазвичай простіший і швидший за обчислення;
- RSA-SHA256 (RS256) – асиметричний алгоритм: для підпису використовується приватний ключ, а для перевірки – відповідний публічний ключ. При цьому система видачі має приватний ключ, а сервіси, що приймають токен, мають публічний ключ. Такий підхід дає додаткову безпеку: лише той, хто володіє приватним ключем, може згенерувати підпис, що і гарантує неможливість його спростувати (non-repudiation). Сервер, який перевіряє підпис, не потребує зберігати секрет – достатньо публічного ключа. В сучасних рекомендаціях віддають перевагу RS256 через підвищену стійкість: якщо приватний ключ буде скомпрометовано, його можна перевипустити без потреби змінювати конфігурацію клієнтів (достатньо оновити публічні ключі). Однак RS256 складніший в обчисленні порівняно з HS256.

Обидва алгоритми забезпечують автентичність токена (перевірку, що дані не змінені і підпис виконано валідним ключем) [7]. Зазвичай рекомендується застосовувати RS256 у продакшені і резервувати HS256 для простих або застарілих середовищ, де асиметрія не підтримується. При цьому важливо регулярно оновлювати (ротувати) ключі, щоб зменшити ризик компрометації, незалежно від вибору алгоритму.

Після створення токен передається клієнту (наприклад, веб-клієнту чи мобільному застосунку). Зазвичай клієнт зберігає отриманий JWT (у пам’яті програми або у сховищі браузера) і додає його до кожного наступного запиту до захищених ресурсів [6]. Згідно зі звичайною практикою, токен поміщається у заголовок HTTP Authorization у формі «Bearer <token>» (рисунок 2.3).

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Рис. 2.3. Вигляд токену у заголовці HTTP Authorization

Далі сервер-ресурс отримує запит з токеном, декодує header та payload і перевіряє цифровий підпис за вказаним алгоритмом. Якщо підпис вірний, далі перевіряються значення стандартних полів, зокрема exp (перевірка, що токен ще не прострочено) та, за потреби, nbf, iat, aud тощо, щоб упевнитися, що токен належить очікуваному випускнику і призначений цьому сервісу. У разі успішної верифікації (підпис правильний і claims коректні) сервер надає доступ до ресурсу на основі інформації з payload – наприклад, видає права відповідно до ролей чи атрибутів користувача.

Кожен JWT має обмежений строк життя, визначений у його payload (через поле exp). Так, токен містить інформацію про час видачі (iat) та час завершення дії (exp). Після спливу exp токен вважається недійсним, і сервер повинен відмовити в доступі. Важливо зауважити, що JWT є токеном типу «Bearer» – той, хто володіє токеном, може ним скористатися. Саме тому поширеною практикою є встановлення невеликого періоду дії (наприклад, кілька хвилин чи годин) і регулярна ротація токенів. Після завершення дії старого токена доступ до ресурсів

неможливий без отримання нового (через систему refresh-токенів або повторну автентифікацію). Додатково деякі системи реалізують списки відмовлених токенів (token revocation), але в базовому випадку JWT самостійно припиняє діяти за наведений час (exp) [5, 7].

2.2 Архітектура та компоненти системи автентифікації на основі JWT

Системи аутентифікації на основі JWT побудовані за принципом безстанового (stateless) відокремлення процесу ідентифікації від бізнес-логіки сервісів. Класична архітектура передбачає виділення окремих ролей: клієнта, що надсилає запити; сервера автентифікації (Identity Provider), що перевіряє облікові дані і видає підписані токени; і ресурсного сервера (API), що захищає дані та верифікує токени. Згідно з рекомендаціями OWASP, аутентифікацію доцільно централізувати в окремому Identity Provider, який видає токени доступу. Такий підхід дозволяє ресурсним сервісам не зберігати стан користувацьких сесій і спрощує масштабування системи (горизонтальне масштабування) [8]. Також є можливість спроектувати систему автентифікації без відокремленого Identity Provider, що значно знижує навантаження на систему в цілому. Однак цей підхід вимагає ретельної реалізації механізмів захисту токенів, так як ротація токенів та збереження секретного ключа треба буде реалізувати с нуля, та проводити ретельне тестування. Типовий цикл аутентифікації із JWT виглядає так [5, 7] (рисунок 2.4):

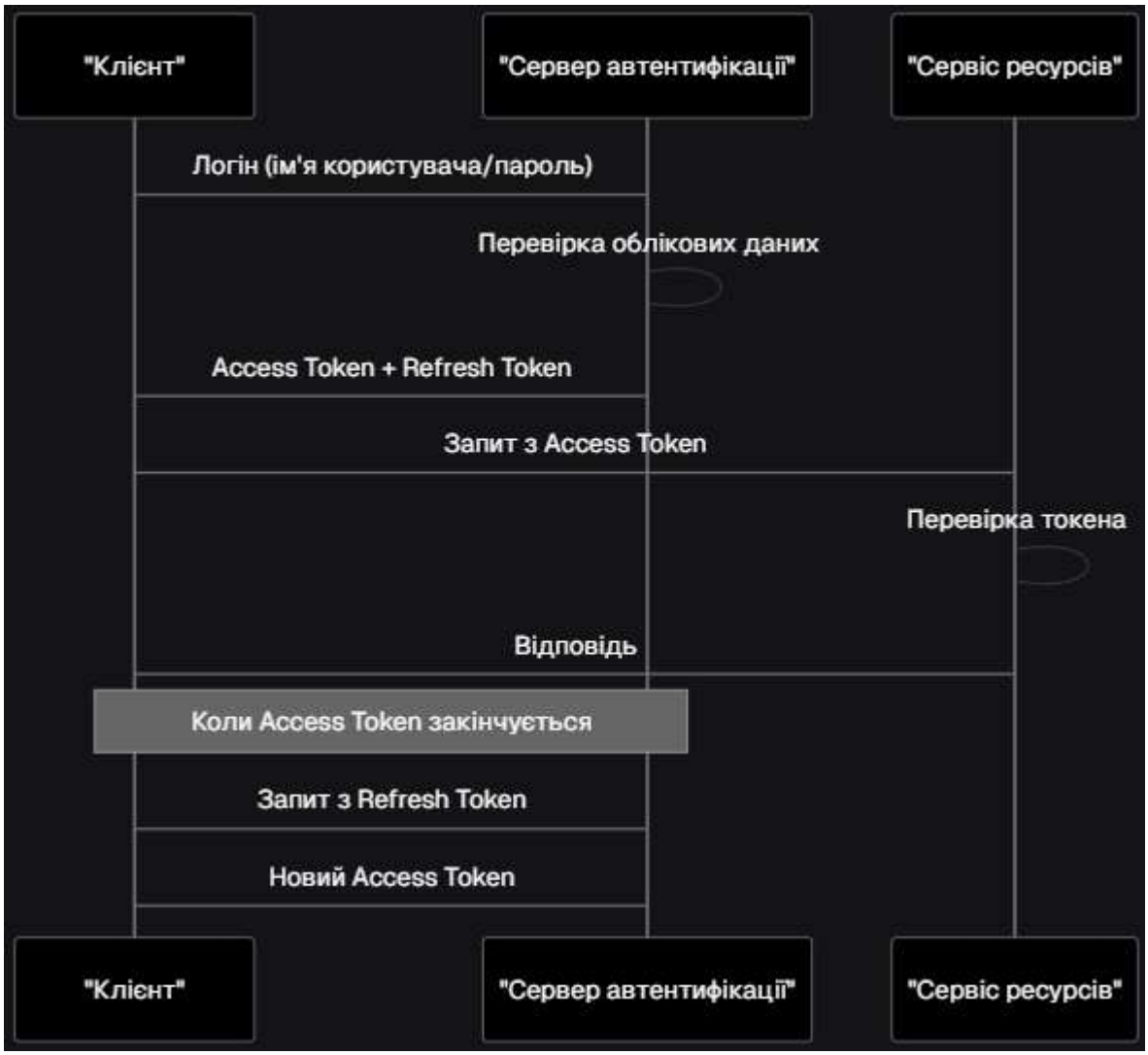

Рис. 2.4. Типовий цикл аутентифікації із JWT

- клієнт надсилає запит на сервер автентифікації з обліковими даними (логін/пароль);
- сервер автентифікації перевіряє надані облікові дані у своїй базі даних.
- якщо дані вірні, сервер автентифікації генерує два токени (access, refresh)
- сервер автентифікації підписує обидва токени за допомогою секретного ключа;
- сервер автентифікації відправляє обидва токени клієнту.
- клієнт зберігає отримані токени (у локальному сховищі або Cookies);
- клієнт надсилає запит до сервісу ресурсів, додаючи access-токен у заголовок Authorization;

- сервіс ресурсів перевіряє підпис access-токена за допомогою того ж секретного ключа;
- сервіс ресурсів перевіряє термін дії токена та інші claims;
- Якщо токен валідний, сервіс ресурсів обробляє запит і надає відповідь клієнту;
- коли термін дії access-токена закінчується, клієнт виявляє це (або при отриманні помилки 401, або превентивно);
- клієнт надсилає запит на сервер автентифікації з refresh-токеном;
- сервер автентифікації перевіряє валідність refresh-токена;
- якщо refresh-токен валідний, сервер автентифікації генерує новий access-токен, та відправляє його клієнту;
- клієнт продовжує використовувати новий access-токен для доступу до ресурсів;
- якщо refresh-токен використовується повторно (що може свідчити про компрометацію), сервер автентифікації анулює всі токени користувача;
- при виході з системи клієнт видаляє токени зі свого сховища;
- сервер автентифікації може додатково вести "чорний список" відкликаних токенів для підвищення безпеки.

Клієнт (Client): програмне забезпечення (наприклад, веб або мобільний додаток), що ініціює запит на аутентифікацію користувача або доступ до ресурсу. Клієнт передає облікові дані серверу автентифікації та отримує назад маркер доступу (JWT).

Сервер автентифікації (Authentication Server / Identity Provider): сервіс, що реалізує протоколи OpenID Connect/OAuth2 і відповідає за перевірку облікових даних користувача. Після успішної аутентифікації цей сервер генерує і підписує JWT. Він же може видавати refresh-токени для продовження сесії. Сервер автентифікації іноді називають сервером авторизації (authorization server) або стейк токенів (security token service) [9].

Генератор JWT (JWT Generator): компонент, що формує і підписує маркери на основі інформації про користувача. Зазвичай це модуль самого сервера автентифікації. Генератор додає до токена необхідні claims (записи про суб'єкт, термін дії, права доступу тощо) та захищає його цифровим підписом.

Ресурсний сервер (API Resource Server): захищений бекенд, який обробляє запити клієнта. При надходженні запиту із JWT він перевіряє підпис і вміст маркера (наприклад, шляхом отримання відкритих ключів з сервера автентифікації) і тільки за успішної валідації надає доступ до запитуваного ресурсу [10, 11]. Ресурсний сервер самостійно оцінює достовірність токена, не звертаючись за цим до сервера автентифікації в реальному часі.

Сховище токенів (опціонально): при роботі з refresh-токенами або для реалізації механізмів відкликання може знадобитися централізоване сховище (база даних). Наприклад, для відкликаних JWT можна використовувати deny-list – таблицю в БД із відміченими ідентифікаторами заблокованих токенів, або redis [12]. Також у сховище можуть зберігатися refresh-токени, щоб контролювати їхнє дійсне життя та відкликати за потреби.

Логування (Logging): ведення журналів подій аутентифікації й авторизації – важливий елемент безпеки. Рекомендується фіксувати як успішні, так і невдалі спроби входу, а також помилки валідації токенів [8]. Аудит таких логів дозволяє виявляти спроби несанкціонованого доступу і відстежувати активність у системі.

Загалом, безстановий підхід, який властивий для JWT надає певні переваги.

Горизонтальна масштабованість та простота розгортання: оскільки сервер не зберігає стан сесії, кластер ресурсних сервісів можна масштабувати без узгодження спільного сховища сесій [8]. Будь-який інстанс ресурсного сервера може перевіряти JWT незалежно, що спрощує балансування навантаження.

Децентралізована валідація: довірені компоненти (ресурсні сервіси) самостійно перевіряють підпис токена, не звертаючись до централізованого сервера авторизації. Це підвищує відмовостійкість системи (відсутня єдина точка відмови) та знижує затримки на запити.

Безпека через криптографію: JWT містять цифровий підпис, тож будь-яка спроба підробити вміст токена призведе до відмови валідації [13]. Крім того, обмеження часу життя токенів і механізми відкликання додатково захищають від несанкціонованого доступу.

Компактність і портативність: формат JWT базується на легковагому JSON, що менше за обсягом порівняно зі складнішими XML-токенами (наприклад, SAML) (див. рис. 2.5) [6].

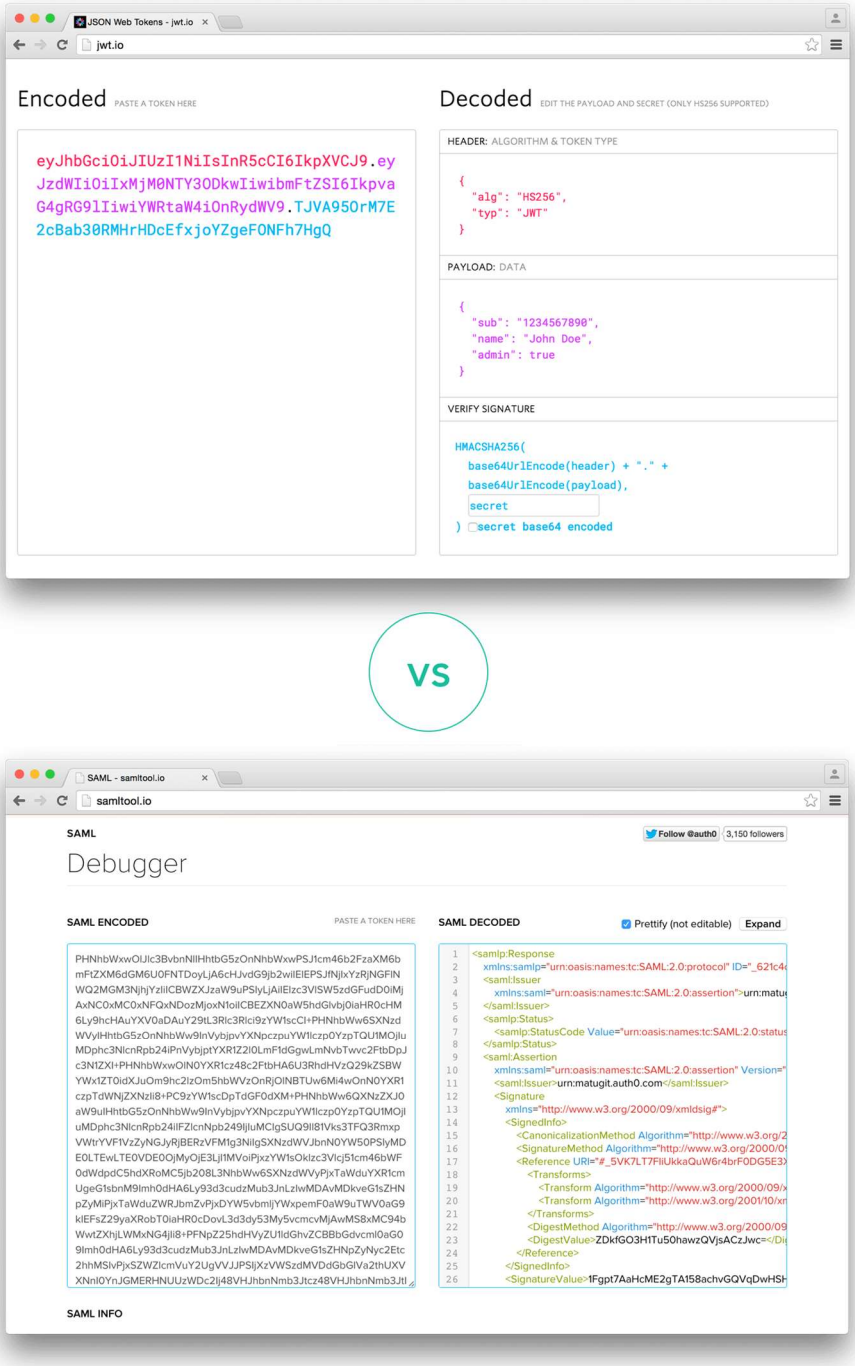

Рис. 2.5. Порівняння JWT та SAML [6]

Це робить передачу маркерів ефективнішою у веб-середовищі. Крім того, JWT є відкритим стандартом і підтримується багатьма платформами, що спрощує інтеграцію між різнорідними сервісами.

Таким чином, архітектура на основі JWT забезпечує гнучкість і продуктивність розподіленої системи автентифікації, дозволяючи централізувати управління ідентифікацією користувачів та одночасно масштабувати ресурси без додаткового ускладнення серверної інфраструктури.

2.3 Основні механізми захисту токенів JWT

Механізми захисту токенів JWT є критично важливою складовою сучасних систем безпеки, оскільки саме вони запобігають несанкціонованому доступу до конфіденційних даних. На відміну від традиційних сесійних механізмів, де стан зберігається на сервері, JWT є самодостатніми токенами, що містять усю необхідну інформацію для автентифікації, що одночасно є їх перевагою та слабким місцем. Основними загрозами для JWT є викрадення токенів, підробка підпису, replay-атаки та несанкціоноване використання вже відкликаних токенів. Для протидії цим загрозам розроблено цілий комплекс заходів захисту, які включають криптографічні методи, правильне налаштування термінів дії токенів, додаткові перевірки та спеціальні архітектурні рішення. Ефективний захист JWT вимагає не лише технічної реалізації, але й розуміння потенційних векторів атак та способів їх нейтралізації. Успішне впровадження цих механізмів дозволяє зберегти всі переваги безстанової автентифікації, мінімізувавши при цьому ризики компрометації системи.

Усі запити й відповіді, що передають JWT, повинні здійснюватися через захищений канал зв’язку (HTTPS/TLS), щоб захистити дані від MITM-атак. Згідно з NIST, токени мають передаватися лише через автентифікований захищений канал

[14]. Це означає використання TLS із дійсним сертифікатом; при цьому слід вимагати серверного сертифіката для аутентифікації кінцевих точок.

Політику Cross-Origin Resource Sharing (CORS) слід обмежити довіреними доменами: якщо токен передається в HTTP-заголовку Authorization, проблеми CORS мінімальні (адже такі запити не включають Cookie) [6]. Для захисту від MITM-атак додатково варто застосовувати HTTP Strict Transport Security (HSTS) – це політика для браузерів яка каже примусово використовувати захищений протокол HTTPS, навіть якщо сервер може приймати і HTTP.

При використанні Cookie слід встановлювати атрибути HttpOnly, Secure і SameSite. HttpOnly унеможливлює доступ JavaScript до Cookie, знижуючи ризик XSS-атаки. Атрибут Secure забезпечує, що токен передається виключно по HTTPS. SameSite обмежує відправлення Cookie лише у контекстах того самого сайту, що значно захищає від CSRF-атак. Наприклад, налаштування виду Set-Cookie: token=…; HttpOnly; Secure; SameSite=Strict; Max-Age=3600 гарантує, що токен не доступний скриптам, надсилається лише по зашифрованому каналу і тільки у запитах на власний домен. Якщо ж JWT зберігається у web storage (localStorage/sessionStorage), це полегшує використання, але створює сильну вразливість до XSS: ін’єкція JavaScript дозволить зловмиснику викрасти токен [16]. Тому при клієнтському зберіганні JWT потрібно максимально уникати передачі токена скриптам і ретельно валідовати вхідні дані.

Токени мають мати якомога коротший строк життя. Параметр exp у JWT задає час закінчення дії; ресурсний сервер або клієнт повинен відхилити токен, якщо він прострочений [14]. Згідно з практикою OAuth 2.0, access-токени зазвичай видаються на короткі періоди (хвилини–години) для мінімізації наслідків витоку. Refresh-токенам можна задавати більший час життя, але навіть вони повинні періодично оновлюватися або відкликатися. У разі використання токенів збережених у токенах доцільно встановлювати атрибут exp, щоб браузер автоматично видаляв токен при закінченні терміна дії [15]. Як зазначено в практиці, короткоживучі токени значно знижують вікно можливого використання викраденого токена.

Для зменшення ризику витоку слід використовувати ротацію refresh-токенів. При кожному обміні refresh-токена на новий access-токен також видається новий refresh-токен, а попередній стає недійсним [16]. Таким чином, немає «довгоживучого» рефреш-токена, і підвищується безпека: якщо токен скомпрометовано, його повторне використання швидко виявиться. За таких схем повинен бути вбудований механізм виявлення повторного використання (reuse detection): якщо старий (вже відкликаний) refresh-токен потрапив у запит, система анулює всі токени даної сесії і вимагає повторної аутентифікації [7]. Крім того, доцільно застосовувати прив'язку до клієнта (client binding) – криптографічно пов'язувати refresh-токен із пристроєм або додатком – щоб викрадені токени не можна було використати з іншого пристрою [16].

Незважаючи на те, що JWT зазвичай використовують у безстанному режимі, у випадках компрометації необхідно можливість негайного відкликання. Для цього впроваджують серверний чорний список (blacklist) або deny list. Ідентифікатори токенів (наприклад, значення jti або ідентифікатор користувача) зберігаються у базі чи кеші (Redis) при відкликанні. Коли користувач надсилає токен, сервер перевіряє, чи немає відповідного jti в чорному списку [17], схема наведена на рисунку 2.6.

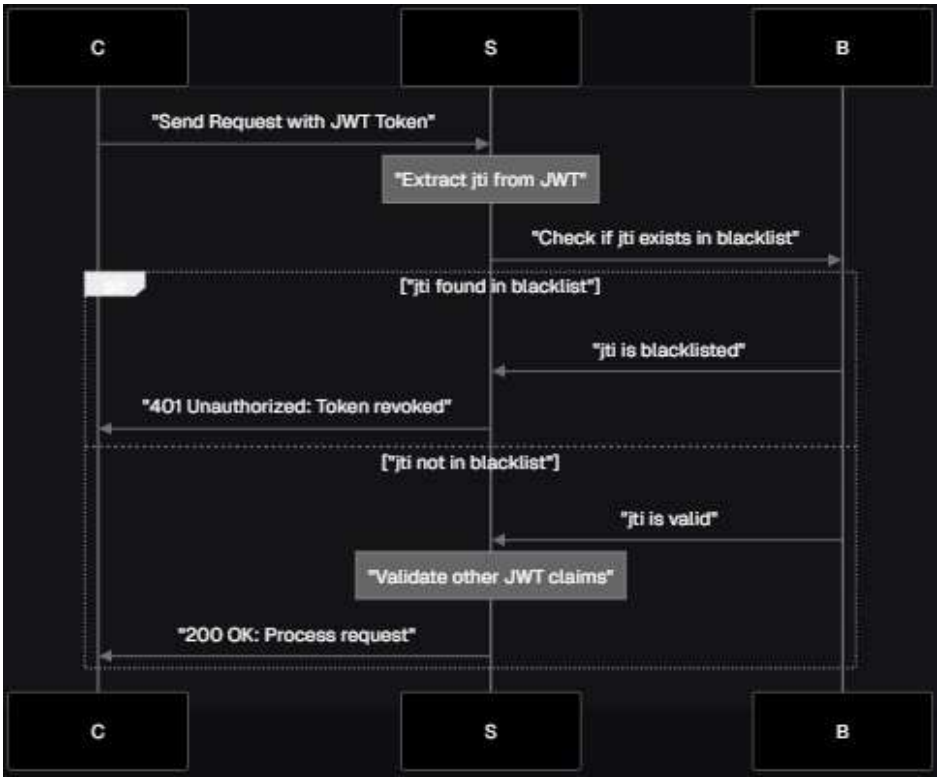

Рис. 2.6. Схема перевірки валідності токена

До переваг такого підходу є відкликання конкретних токенів без впливу на інші сеанси, але необхідне зберігання стану на сервері (суперечить безстатевому підходу) і збільшує навантаження на перевірки.

Наприклад, у Flask можна при відкликанні встановлювати в Redis пару ключ-значення: {"jti": "revoked"}, а при обробці запитів – перевіряти наявність ключа [17]. Такий механізм забезпечує недопущення використання токенів після їх відкликання, навіть якщо вони ще не є простроченими. Також можливий варіант deny list на рівні користувача: зберігати останню зміну облікового запису (наприклад, при зміні пароля чи відключенні сесії), і відхиляти всі токени із часом випуску до цієї події.

JSON Object Signing and Encryption (JOSE) – це набір стандартів і відповідних реалізацій, що забезпечують: підпис (JSON Web Signature – JWS); шифрування (JSON Web Encryption – JWE); обробку ключів (JSON Web Key – JWK, JSON Web Algorithms – JWA); ідентифікатори (JWT як специфічна реалізація з використанням JWS або JWE).

Згідно з RFC 8725, бібліотеки JOSE повинні дозволяти застосування тільки очікуваних алгоритмів: розробник повинен явно вказувати, які алгоритми допустимі, а бібліотека не має приймати інші. Необхідно суворо перевіряти, що заголовок alg у JWT співпадає з реальним алгоритмом перевірки, та що один ключ використовується лише для одного алгоритму. Це запобігає класу атак, коли зловмисник змінює alg на «none» або підміняє алгоритм із асиметричного на симетричний. У RFC 8725 прямо згадано: «зловмисник може змінити алгоритм на «none», і деякі бібліотеки повірять цьому значенню та “перевірять” JWT без перевірки підпису». Аналогічний ризик – «конфузія» між RS256 і HS256: якщо сервер очікує RSA-підпис (RS256), а отримує HMAC (HS256), бібліотека може трактувати відкритий RSA-ключ як HMAC-секрет, що дозволяє зловмиснику підписувати токени своїм знанням відкритого ключа. Тому забороняється використовувати {"alg":"none"} без явної потреби. JWT-бібліотеки не повинні видавати або приймати токени з {"alg":"none"}, якщо це не навмисне рішення для нетравмованої роботи (наприклад, при додатковому TLS-захисті). При виборі

алгоритмів рекомендується віддавати перевагу сучасним стандартам: HMAC з довгим секретом (при симетричному підписуванні) або RSA/ECDSA з достатньою довжиною ключа (наприклад, RSA 2048/3072 бит). Важливо правильно зберігати і обертати ключі: симетричний HMAC-секрет має бути випадковим і достатньо довгим, а приватні RSA-/ECDSA-ключі – захищені від несанкціонованого доступу (наприклад, у Hardware Security Module (HSM) чи захищеному сховищі). У жодному разі не слід передавати секретний ключ клієнту – при асиметричному підписуванні поширюють лише публічний ключ, що знижує ризик викрадення [18].

Для повноти захисту критично важливо реєструвати (логувати) всі події, пов'язані з аутентифікацією та авторизацією. OWASP Logging Cheat Sheet рекомендує вести журнали для усіх безпекових подій і невдалих спроб доступу [19]. Сюди належать: видача нового токена; невдала валідація (помилки підпису, невірні aud/iss/exp); використання черного списку; ротація токенів; підозрілі відмови в доступі. Регулярний аудит таких логів допомагає виявити атакуючі сценарії й аналітично оцінити безпеку системи.

Більші організації і стандарти (NIST, ENISA, Google) також рекомендують дотримуватися відомих «best practices». Наприклад, NIST SP 800-63B наголошує, що access-токен не означає факт активної сесії користувача і може бути довгостроково дійсним без підтвердження присутності користувача, тому для критичних систем слід вимагати періодичної повторної аутентифікації [1]. ENISA та Google Cloud рекомендують використовувати принцип найменших привілеїв (скоуп токена має бути вузьким) і не зберігати чутливих даних у невбитому вигляді всередині JWT. Згідно з OWASP JWT Cheat Sheet, всі «критичні атаки» (підробка заголовку alg, повторне використання, витік через XSS/CSRF) мають бути враховані, а застосунок повинен «не довіряти вхідним claim» без перевірки [18].

У сукупності, лише скрупульозне дотримання перерахованих механізмів (TLS, атрибути безпеки, короткий строк дії, ротація/ревокація, верифікація claims, суворий контроль алгоритмів) і рекомендацій провідних організацій забезпечить належний рівень захисту JWT у сучасних системах аутентифікації та авторизації.

3 РОЗРОБКА РЕКОМЕНДАЦІЙ ЩОДО ЗАСТОСУВАННЯ JWT У СИСТЕМАХ АВТЕНТИФІКАЦІЇ

3.1 Реалізація системи безпечної автентифікації на основі JWT у Flask додатку

Реалізація безпечної системи автентифікації на основі JWT не обмежується лише логікою бекенду – вона повинна враховувати весь шлях запиту від клієнта до вебзастосунку. Хоча сам токен використовується для забезпечення безперервної аутентифікації та авторизації, безпечна обробка запитів починається ще до надходження їх у Flask-додаток. Саме на цьому етапі важливо впроваджувати захисні механізми, які протистоять типу атак, що можуть скомпрометувати навіть правильно реалізовану JWT-систему.

Типова архітектура для Web Server Gateway Interface (WSGI) фреймворків виглядає наступним чином: клієнт надсилає запит до веб додатку; запит йде до проксі серверу (найбільш завжди це Cloudflare); запит перенаправляється до сервера де розміщений сам веб застосунок; запит перехоплюється зворотним проксі (Reverse Proxy) (найчастіше це Nginx або Apache); запит йде до WSGI сервера; запит перенаправляється до Flask додатку (див. рис. 3.1).

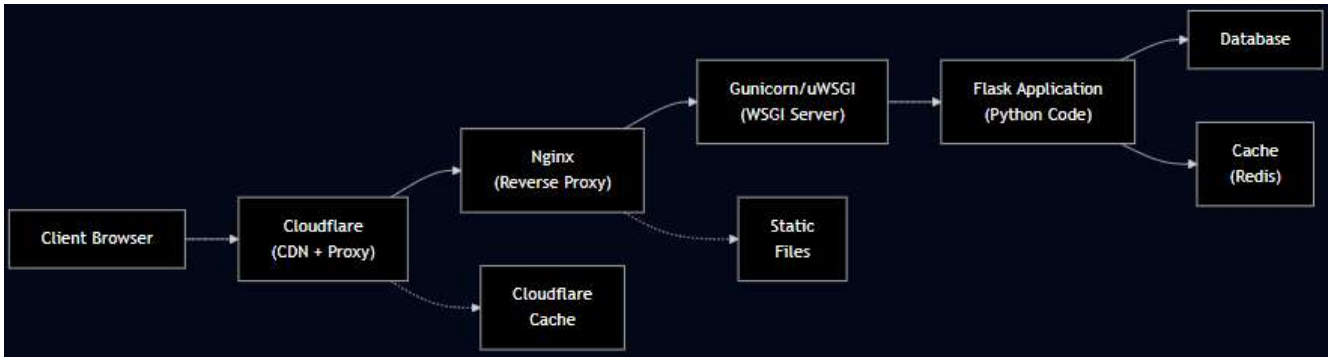

Рис. 3.1. Типова архітектура WSGI

Для того щоб гарантувати безпечне використання JWT, необхідно врахувати можливі атаки на кожному з цих рівнів – від мережевого перехоплення до CSRF, XSS та Replay-атак. Тому перед безпосередньою реалізацією механізмів аутентифікації важливо розглянути засоби захисту, що впроваджуються на рівні інфраструктури – зокрема, через конфігурацію Cloudflare та зворотного проксі-сервера Nginx.

Cloudflare забезпечує глобальну мережу з більш ніж 275 дата-центрами, що означає мінімальну затримку для користувачів з будь-якої точки світу. Контент кешується на edge серверах (сервери, які розміщуються на межі мережі та обробляють дані ближче до джерел даних або кінцевим користувачам), зменшуючи навантаження на основний сервер та прискорюючи відгук для повторних запитів. Блокує DDoS атаки, ботів та шкідливий трафік ще до того, як він досягне сервера. Автоматично надає та оновлює SSL сертифікати, забезпечуючи HTTPS з’єднання без додаткових налаштувань. Можна налаштувати end-to-end шифрування між Cloudflare та вашим сервером. Cloudflare надає детальну аналітику трафіку, атак та продуктивності, забезпечує високу доступність навіть якщо основний сервер тимчасово недоступний.

Nginx додає додатковий рівень фільтрації та може обмежувати швидкість запитів, логує всі запити для подальшого аналізу. Це дозволяє відстежувати проблеми та оптимізувати роботу, це створює потужний захисний бар’єр. Nginx ефективно обслуговує статичні файли (CSS, JS, зображення) без залучення Python процесів, а Cloudflare кешує їх глобально. Це значно зменшує навантаження на Flask додаток.

Gunicorn може запускати кілька worker процесів, обробляючи паралельні запити. Nginx може балансувати навантаження між кількома екземплярами Gunicorn.

Зменшення навантаження на основний сервер через кешування дозволяє використовувати менш потужне обладнання. Cloudflare має безкоштовний план з базовими функціями.

Така структура має багато переваг з точки зору оптимізації навантаження, а також дуже сильно підвищує надійність та відмовостійкість Flask додатку. Cloudflare захистить додаток від атак типу DDoS, та замаскує реальний IP сервера, а від Brute-force та інших атак вже захистить Nginx.

Nginx може обмежувати кількість запитів з одного IP адреса (Rate Limiting) через модуль limit_req. Це особливо критично для захисту форм логіну, API endpoints та адміністративних панелей (див. рис. 3.2).

```
# Обмеження до 5 запитів на хвилину для логіну
limit_req_zone $binary_remote_addr zone=login:10m rate=5r/m;

location /login {
    limit_req zone=login burst=3 nodelay;
    proxy_pass http://flask_backend;
}
```

Рис. 3.2. Реалізація Rate Limiting

Nginx може блокувати трафік з певних країн або регіонів, що часто є джерелом автоматизованих атак. Використовуючи GeoIP модуль, можна створити whitelist або blacklist країн (див. рис. 3.3).

```
# Блокування трафіку з певних країн
if ($geoip_country_code ~ (CN|RU|KP)) {
    return 403;
}
```

Рис. 3.3. Реалізація географічної фільтрації

Nginx може фільтрувати запити на основі підозрілих User-Agent (рядок, який клієнтська програма, наприклад, веб-браузер, відправляє веб-серверу разом із запитом) заголовків, порожніх referrer (HTTP-заголовок, який містить URL-адресу сторінки, з якої користувач перейшов на поточну сторінку) або відомих сигнатур ботів. Це ускладнює атаки типу Brute-Force, оскільки зловмисник буде зобов’язаний підставляти до кожного запиту не заблокований User-Agent та referrer (див. рис. 3.4).

```
# Блокування підозрілих User-Agent
if ($http_user_agent ~* (bot|crawler|spider|scraper)) {
    return 403;
}

# Блокування порожніх referrer для певних endpoints
if ($http_referer = "") {
    return 403;
}
```

Рис. 3.4. Реалізація захисту від User-Agent та Referrer атак

Nginx може захистити від атак, які намагаються виснажити сервер через повільні з'єднання (Slowloris) (див. рис. 3.5).

```
# Таймаути для захисту від повільних атак
client_body_timeout 10s;
client_header_timeout 10s;
keepalive_timeout 5s 5s;
send_timeout 10s;
```

Рис. 3.5. Реалізація захисту від атак типу Slowloris

Nginx дозволяє створювати списки дозволених (Whitelisting) та заборонених (Blacklisting) IP адрес, що особливо корисно для адміністративних розділів. Також завдяки цьому методу можна блокувати користувачів, які користуються сервісами VPN. Це ускладнює атаки типу Brute-Force, оскільки зловмиснику знадобляться списки незаблокованих проксі серверів, це робить атаки цього типу доволі дорогими (див. рис. 3.6).

```
# Дозвіл доступу тільки з офісних IP
location /admin {
    allow 192.168.1.0/24;
    allow 10.0.0.0/8;
    deny all;
    proxy_pass http://flask_backend;
}
```

Рис. 3.6. Реалізація IP Whitelisting та Blacklisting

Nginx логи можуть використовуватися з Fail2Ban для автоматичного блокування IP адрес після певної кількості невдалих спроб. Це ускладнює атаки

типу Brute-Force, оскільки IP зловмисника буде заблоковано через декілька повторних запитів, знадобляться списки незаблокованих проксі серверів, це робить атаки цього типу доволі дорогими (див. рис. 3.7).

```
# Fail2Ban правило для Nginx
[nginx-limit-req]
enabled = true
filter = nginx-limit-req
action = iptables-multiport[name=ReqLimit, port="http,https", protocol=tcp]
logpath = /var/log/nginx/error.log
findtime = 600
bantime = 7200
maxretry = 10
```

Рис. 3.7. Реалізація Fail2Ban

Варто зауважити, що значну частину загроз безпеки можна нейтралізувати ще на рівні веб-сервера завдяки правильному налаштуванню зворотного проксі, зокрема Nginx. Проте, ручне створення повноцінного захисного конфігураційного файлу, який охоплював би всі актуальні вектори атак, є складним, об’ємним і ресурсоємним завданням.

З метою спрощення та стандартизації цього процесу активно використовується OWASP ModSecurity Core Rule Set (CRS) – набір правил для веб-аплікаційного фаєрвола (WAF), який охоплює найпоширеніші типи атак згідно з класифікацією OWASP Top 10. CRS забезпечує захист від таких загроз, як SQL-ін’єкції, XSS, Local File Inclusion, Command Injection, та інші типи шкідливих запитів. Інтегруючи ModSecurity з Nginx і підключивши CRS, можна значно підвищити загальну стійкість застосунку до типових атак ще до обробки запиту у Flask-додатку (див. рис. 3.8 – 3.10).

```
# Клонування останньої версії правил
git clone https://github.com/coreruleset/coreruleset.git
cd coreruleset
```

Рис. 3.8. Клонування останньої версії правил CRS

```
#!/bin/bash
# Скрипт для автоматичного оновлення правил
cd /etc/nginx/modsecurity/coreruleset
git pull origin v3.3/master
nginx -t && systemctl reload nginx
```

Рис. 3.9. Скрипт для автоматичного оновлення правил

```
# Основна конфігурація ModSecurity
load_module modules/ngx_http_modsecurity_module.so;

http {
    modsecurity on;
    modsecurity_rules_file /etc/nginx/modsecurity/main.conf;
}

server {
    location / {
        modsecurity_rules '
            Include /etc/nginx/modsecurity/coreruleset/crs-setup.conf
            Include /etc/nginx/modsecurity/coreruleset/rules/*.conf
        ';
        proxy_pass http://flask_backend;
    }
}
```

Рис. 3.10. Основна конфігурація ModSecurity

Усе вищезазначене забезпечує базову захищеність на рівні мережевої інфраструктури, але основна логіка контролю доступу та автентифікації реалізується всередині самого Flask-додатку. Для реалізації надійної системи автентифікації доцільно використовувати JWT з архітектурою access та refresh токенів, що забезпечує оптимальний баланс між безпекою та зручністю використання. Центральним елементом такої системи є єдиний endpoint «/auth», який обробляє всі операції автентифікації через різні HTTP методи: GET для процесу логіну, POST для реєстрації нових користувачів, PATCH для оновлення токенів та DELETE для логауту. Така консолідована архітектура спрощує маршрутизацію, полегшує налаштування rate limiting на рівні Nginx та забезпечує централізоване логування всіх операцій автентифікації (див. рис. 3.11).

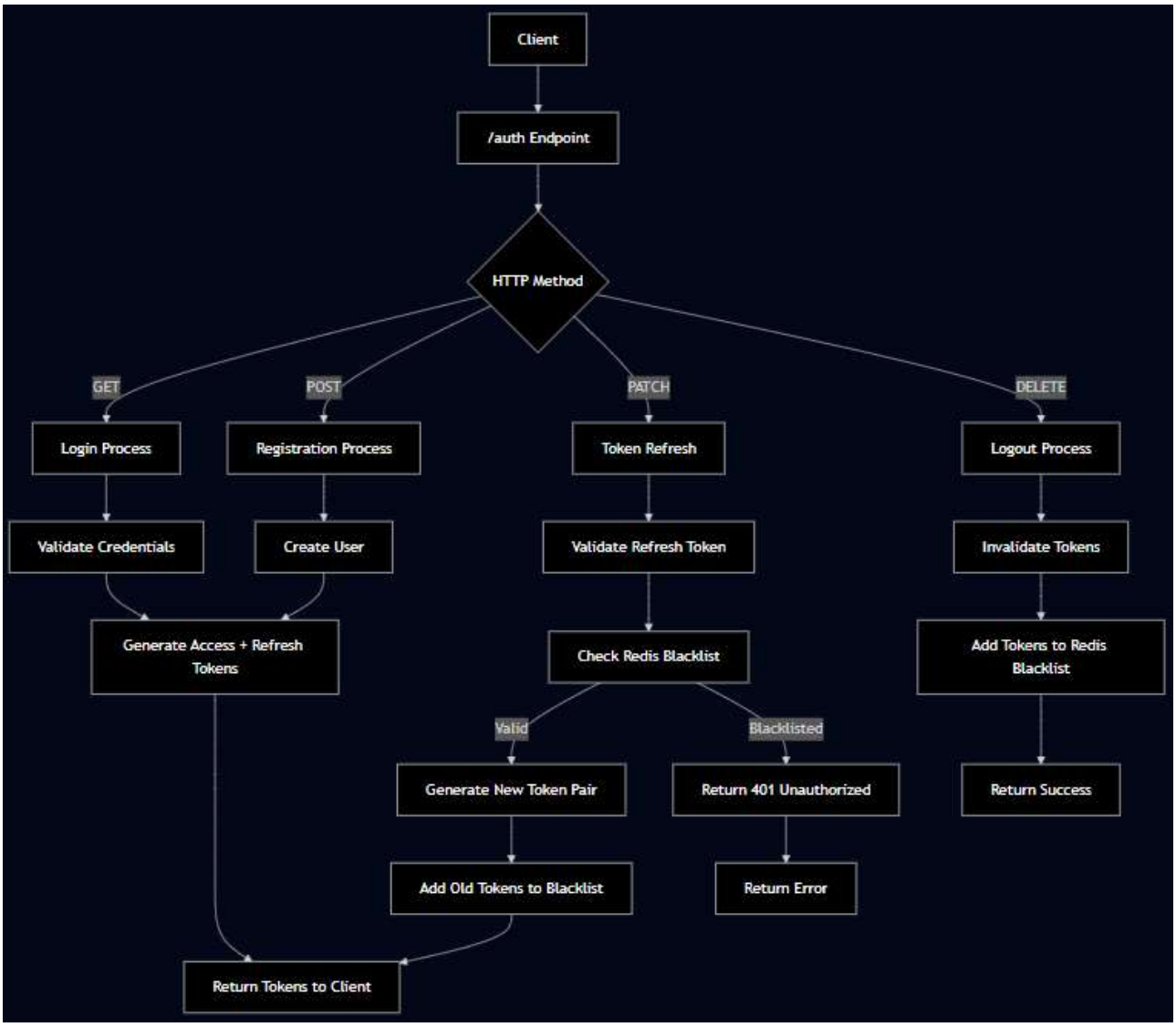

Рис. 3.11. Запропонована архітектура безпечної автентифікації на основі JWT

Треба зазначити, що усі данні, які приходять з клієнта (облікові данні, будь які данні з форм додатка та інші) особливо при запитах типу POST можуть вміщувати SQL-ін’єкції або інший шкідливий зміст. Якщо використовується ModSecurity WAF з CRS атаки такого типу майже сто відсотково будуть заблоковані, але це не є підставою не робити жорстку валідацію вхідних даних у додатку. Один з варіантів як це можна зробити – регулярні вирази (regexp) для кожного поля окремо (див. рис. 3.12).

```
email_regexp = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
"""
This regular expression validates email addresses.
```

Рис. 3.12. Регулярний вираз для пошти

Процес логіну ініціюється GET запитом до /auth endpoint з передачею облікових даних користувача через заголовки Authorization або JSON тіло запиту (див. рис. 3.13).

Рис. 3.13. Передача облікових даних через JSON тіло запиту

Сервер отримує email та password, після чого виконує валідацію через порівняння з хешованим паролем у базі даних, використовуючи модуль bcrypt або flask-bcrypt (див. рис. 3.14).

```
user = UserDAO.get_user_by_email(email)
if user is None or not bcrypt.check_password_hash(user.password, password):
    return jsonify({'err': 'Unauthorized', 'msg': 'Email or password is incorrect'}), 401
```

Рис. 3.14. Перевірка дійсності автентифікаційних даних через порівняння з хешованим паролем у базі даних

При успішній валідації система генерує пару токенів завдяки модулю flask_jwt_extended (див. рис. 3.15).

```
refresh_token = create_refresh_token(identity=user.uuid)
access_token = create_access_token(
                  identity=user.uuid,
                  fresh=True,
                  additional_claims={'refresh_jti': decode_token(refresh_token)['jti']})
```

Рис. 3.15. Генерація пари токенів

Довготерміновий refresh token з identity (user_uuid) та значно підвищеним терміном закінчення (24 години).

Як identity використовується не звичайний ID, а Universally Unique Identifier (UUID). Це особливо важливо у базах даних, де за замовчуванням використовується звичайний інкремент для генерації наступного унікального ідентифікатора у таблиці (такі як SQLite3, MySQL, PostgreSQL та інші), у таких базах даних як MongoDB за замовчуванням використовуються UUID замість ID. Якщо у JWT або URL фігурують прості числові ID, зловмисник може автоматизовано перебирати їх і таким чином відтворювати структуру БД – дізнаватись, скільки існує користувачів, які з них активні, які мають певні ролі чи властивості. UUID унеможливлює таку індексацію: він не передбачуваний, не дає інформації про кількість записів, і не дозволяє «вивести» логіку росту ID.

Короткотерміновий access token з identity, роллю користувача (опційно, якщо треба реалізувати Role-Based Access Control (RBAC)), часом закінчення дії (15 хвилин), поміткою «fresh=True» та refresh_jti (унікальний ID створеного refresh токена).

Помітка «fresh=True» відноситься до специфіки модуля flask_jwt_extended та вказує на те, що access-токен був створений після проходження автентифікації, а не завдяки refresh-токену. Це важливий момент, оскільки у веб застосунку зазвичай є чуткі роути (наприклад зміна паролю, або видалення акаунту), для доступу до яких краще бути впевненим, що це саме користувач, а не зловмисник викравший токени доступу.

Вказувати ID створеного refresh токена не обов’язково але це спрощує відкликання токенів при логауті користувача, не треба передавати обидва токена, достатньо лише access-токена.

Обидва токени підписуються секретним ключем сервера (робиться автоматично завдяки модулю flask_jwt_extended) та повертаються клієнту у httpOnly Cookie (див. рис. 3.16).

```
resp = jsonify({'msg': 'Successful login'})
set_access_cookies(resp, access_token)
set_refresh_cookies(resp, refresh_token)
return resp, 200
```

Рис. 3.16. Встановлення Cookie

У методах створення Cookie «set_acccess_cookies» та «set_refresh_cookies» за замовчуванням встановлений «httponly=True» (Cookie недоступні з JavaScript (захист від XSS-атак)) отже його не треба вказувати окремо. Також краще вказати використання Cookie тільки з HTTPS та заборонити міжсайтовий обмін Cookie (див. рис. 3.17).

```
app.config['JWT_COOKIE_SECURE'] = True  # Куки передаються лише через HTTPS
app.config['JWT_COOKIE_SAMESITE'] = 'Lax'  # Або 'Strict' (обмеження міжсайтових запитів)
```

Рис. 3.17. Налаштування конфігурації створення Cookie

Додатково система може логувати успішний логін з IP адресою, User-Agent та часовою міткою для подальшого аудиту безпеки, але це краще робити завдяки nginx, як було описано раніше.

Реєстрація нового користувача через POST запит включає валідацію наданих даних на унікальність email, складність пароля та відповідність іншим бізнес-правилам додатку (у даному випадку: ім’я, фамілія, пошта та пароль) (див. рис. 3.18).

Рис. 3.18. Передача облікових даних через JSON тіло запиту

Система перевіряє чи не існує вже користувач з таким email у базі даних, хешує пароль за допомогою модуля bcrypt з використанням salt для захисту від атак типу райдужна таблиця (rainbow table), створює новий запис користувача з базовими правами доступу (якщо потрібен RBAC) (див. рис. 3.19).

```
if UserDAO.get_user_by_email(email) is not None:
    return jsonify({'err': 'Conflict', 'msg': 'Email already exists!'}), 409

UserDAO.create_user(first_name, last_name, email, bcrypt.generate_password_hash(password))
```

Рис. 3.19. Реєстрація нового користувача

Після цього одразу генерує пару JWT токенів для негайного автоматичного логіну після реєстрації (див. рис. 3.20).

```
refresh_token = create_refresh_token(identity=user.uuid)
access_token = create_access_token(
                    identity=user.uuid,
                    fresh=True,
                    additional_claims={'refresh_jti': decode_token(refresh_token)['jti']})
```

Рис. 3.20. Генерація пари токенів

Цей підхід покращує користувацький досвід, оскільки новий користувач одразу отримує доступ до системи без необхідності окремого логіну.

Система також може надсилати welcome email або виконувати інші post-registration дії, зберігаючи інформацію про реєстрацію в логах для аналітики та безпеки.

Redis blacklist функціонує як централізоване сховище недійсних токенів, використовуючи TTL механізм для автоматичного очищення застарілих записів. Кожен токен зберігається з ключем що містить його JTI (JWT ID) або хеш, та Time To Live (TTL) що відповідає часу до природного закінчення токена, забезпечуючи оптимальне використання пам’яті. Система перевіряє blacklist при кожному PATCH запиті та може додатково перевіряти access токени при критичних операціях для максимальної безпеки. Redis також може зберігати метадані про причину додавання токена до blacklist (logout, refresh, security breach), що корисно для аналізу безпеки та форензики. Архітектура підтримує горизонтальне масштабування через Redis Cluster та забезпечує високу доступність через replication, критично важливу для додатків з високим навантаженням.

Процес оновлення токенів є критично важливим для підтримання безперервної автентифікації без примушування користувача до повторного логіну.

Коли клієнт надсилає будь який запит у додаток та отримує помилку «access token expired» - це свідчить, що access-токен є простроченим. Після цього клієнт надсилає PATCH запит, додаток витягує refresh-токен з Cookie, перевіряє його наявність у Redis blacklist, за допомогою модуля redis, для виявлення скомпрометованих або анульованих токенів (див. рис. 3.21).

```
if redis_client.exists(get_jwt()["refresh_jti"]):
    return jsonify({"err": "Токен відкликано!"}), 401
```

Рис. 3.21. Перевірка refresh-токена на дійсність

Якщо refresh token відсутній у blacklist, сервер валідує його підпис та термін дії, після чого заносить стару пару у blacklist (access-токен на 15хвилин, refresh-токен на 24 години), генерує нову пару access та refresh токенів з оновленими часовими мітками, та оновлює Cookie клієнта (див. рис. 3.22).

```
new_refresh_token = create_refresh_token(identity=user.uuid)
new_access_token = create_access_token(
    identity=identity=user.uuid,
    fresh=False,
    additional_claims={'refresh_jti': decode_token(new_refresh_token)['jti']})

resp = jsonify({'msg': 'Токен оновлено!'})
set_access_cookies(resp, new_refresh_token)
set_refresh_cookies(resp, new_access_token)
return resp, 200
```

Рис. 3.22. Ротація токенів

Важлива різниця з GET запитом у тому, що access-токен створюється з поміткою «fresh=False», з цим токеном вже не вийде звертатися до чутких роутів. Також критично важливо, що старі токени негайно додаються до Redis blacklist з терміном життя (Time To Live, TTL) що відповідає їх оригінальному терміну дії, запобігаючи можливості replay атак. Нова пара токенів додається до Cookie замість старої, забезпечуючи безшовне продовження сесії. Цей механізм також дозволяє реалізувати token rotation - безпекову практику регулярної зміни токенів для мінімізації ризиків при їх компрометації.

Логаут через DELETE запит забезпечує безпечне завершення користувацької сесії шляхом анулювання всіх активних токенів. Клієнт надсилає запит, додаток витягує access-токен з Cookie, додає до Redis blacklist з відповідними TTL значеннями обидва токена (jti_id refresh-токена знаходиться у payload access-токена) (див. рис. 3.23).

```
redis_client.set(get_jwt()["jti"], 'access token revoked',
                 ex=app.config.get('JWT_ACCESS_TOKEN_EXPIRES'))
redis_client.set(get_jwt()["refresh_jti"], 'refresh token revoked',
                 ex=app.config.get('JWT_REFRESH_TOKEN_EXPIRES'))
resp = jsonify({'msg': 'Tokens successfully revoked'})
unset_jwt_cookies(resp)
return resp, 200
```

Рис. 3.23. Додавання токенів до blacklist, та чистка Cookie

Цей процес гарантує, що навіть якщо токени були скопійовані або перехоплені, вони не можуть бути використані після логауту користувача. Додаток може записувати подію логауту з деталями сесії для аудиту безпеки.

Представлена архітектура безпечної автентифікації на основі JWT у Flask-додатку демонструє сучасний підхід до побудови безпечних веб-застосунків з багаторівневим захистом та надійною системою автентифікації. Комбінація Cloudflare як глобального CDN з можливостю DDoS захисту, Nginx з інтегрованим WAF + CRS для захисту і логування та Flask як основного фреймворку створює стійку до навантажень та атак інфраструктуру. Централізована система автентифікації через єдиний /auth endpoint з JWT токенами та Redis blacklist забезпечує гнучке управління сесіями, підтримку багатопристроєвого доступу та можливість миттєвого анулювання скомпрометованих токенів. Така архітектура масштабується горизонтально, підтримує мікросервісний підхід через стандартизований JWT механізм та забезпечує захист від основних векторів атак включаючи OWASP Top 10. Використання централізованих правил безпеки через ModSecurity CRS та автоматичне оновлення з GitHub репозиторіїв гарантує актуальність захисту без постійного ручного втручання. Результатом є production-ready рішення, що поєднує високу продуктивність, надійність та безпеку, придатне для застосунків з високими вимогами до доступності та захисту даних.

3.2 Виявлення та реагування на атаки, пов’язані з JWT

JWT є популярним засобом автентифікації та авторизації в сучасних веб-додатках. Однак, як і будь-який інший механізм безпеки, JWT піддається певним загрозам, серед яких: повторне використання токенів, маніпуляція підписом, вразливості в алгоритмах підпису, атаки через компрометацію секретного ключа тощо. Тому важливим аспектом розробки системи автентифікації є здатність виявляти такі атаки та оперативно на них реагувати.

Одним із основних інструментів для виявлення аномалій є моніторинг логів аутентифікації. Зокрема, система має відстежувати:

- часті невдалі спроби авторизації;
- повторне використання одного й того ж JWT з різних IP-адрес або пристроїв;
- використання прострочених або відкликаних токенів;
- невідповідність між вмістом токена (наприклад, iss, aud, exp) та поведінкою користувача;

З проблемою частих невдалих спроб авторизації легко справиться Nginx + Fail2Ban. Nginx у даному випадку потрібен для логування спроб авторизації (див. рис. 3.24).

```
log_format security_log '$remote_addr - [$time_local] '
                        '"$request" $status '
                        '"$http_user_agent"';

access_log /var/log/nginx/auth_access.log security_log;
```

Рис. 3.24. Логування спроб авторизації у Nginx-лог

Логи також може робити сам додаток, але якщо у ньому станеться збій, логи можуть не завжди записатися, особливо, якщо їх робить не окремий сервіс, тому краще збирати основні логи через Nginx. Він теж може зазнати збій, але це дуже малоймовірно, тому що Nginx спроектований для працездатності при великих навантаженнях, навіть якщо він перестане працювати у наслідок збою, це не пошкодить додаток, усі запити клієнтів просто не пройдуть або якщо є

балансувальник навантаження, запити перекинуться на інший сервер. Fail2Ban аналізує логи створенні Nginx та шукає ті, у яких статус код більш ніж 400, найчастіше 403 або 401 (див. рис. 3.25 – 3.26).

```
[nginx-auth]
enabled  = true
filter   = nginx-auth
logpath  = /var/log/nginx/auth_access.log
maxretry = 5
findtime = 300
bantime  = 3600
action   = iptables[name=HTTP, port=http, protocol=tcp]
```

Рис. 3.25. Файл конфігу Fail2Ban

```
[Definition]
failregex = ^<HOST> - \[.*\] "GET /auth.*" 401
            ^<HOST> - \[.*\] "GET /auth.*" 403
ignoreregex =
```

Рис. 3.26. Фільтр для виявлення спроб авторизації з кодами 401 або 403

У даному випадку при 5 невдалих спробах авторизації за п’ять хвилин (300 секунд), IP з якого здійснювались запити буде заблоковано на одну годину (3600 секунд). Також якщо використовується SIEM система, можна зробити репорт цієї події.

Для протидії повторного використання одного й того ж JWT з різних IP-адрес можна вшивати IP адресу клієнта, котрому було видано токен у його payload. На сервері тим часом перевіряти IP адреси клієнта та токена, якщо вони не співпадатимуть, відзивати пару токенів (додавати у Redis blacklist) та наказати клієнту знову пройти автентифікацію (див. рис. 3.27).

```
token_ip = get_jwt().get("ip")
current_ip = request.remote_addr

if token_ip != current_ip:
    # Додаємо токен у чорний список
```

Рис. 3.27. Перевірка співпадіння IP токена та клієнта

Для протидії використання прострочених або відкликаних токенів найчастіше використовують Redis blacklist. Якщо э потреба відізвати токен до закінчення TTL, просто додавати його у blacklist та перевіряти токени які приходять із запитом клієнта, якщо токени співпадають – блокувати запит та наказати клієнту повторно авторизуватись.

Зберігання JWT токенів у httpOnly cookies забезпечує захист від XSS атак, але потребує специфічного моніторингу для виявлення CSRF та session fixation атак. Nginx логує всі спроби доступу до /auth endpoint без відповідних CSRF токенів або з підозрілими Referer заголовками, що може свідчити про cross-site атаки. Система відстежує аномалії у cookie поведінці, такі як раптові зміни у cookie значеннях без відповідних refresh операцій, множинні одночасні сесії з різних географічних локацій, або спроби встановлення cookies через JavaScript що може свідчити про XSS атаки. ModSecurity правила перевіряють цілісність cookie структури, виявляючи спроби cookie poisoning або session fixation через аналіз cookie attributes та їх відповідність безпековим стандартам.

Усі компоненти системи потребують генерувати структуровані логи у JSON форматі для централізованого збору через ELK stack (Elasticsearch, Kibana, Beats, Logstash) або подібні рішення, що дозволяє проводити комплексний аналіз інцидентів та виявляти складні багатоетапні атаки. Система зберігає детальну інформацію про кожен JWT токен включаючи його lifecycle, IP адреси з яких він використовувався, User-Agent strings та географічну інформацію для побудови timeline атак та identification зловмисників. Автоматичні звіти повинні генеруватися щоденно з аналізом трендів безпеки, топ атакуючих IP адрес, найчастіших типів JWT атак та ефективності захисних механізмів, що дозволяє оцінити безпекову позицію системи та адаптувати захисні заходи до нових загроз.

3.3 Практичні рекомендації щодо безпечного використання JWT у системах автентифікації

У цьому розділі розглядаються практичні рекомендації, які допоможуть підвищити рівень безпеки під час використання JWT у системах автентифікації. Запропоновані заходи охоплюють аспекти криптографії, збереження та передачі токенів, захисту від компрометації, а також методи реагування на підозрілу активність. Вони є актуальними для розробників, які прагнуть забезпечити цілісність, конфіденційність та автентичність користувацьких сесій.

Завжди треба використовувати HTTPS для передачі JWT токенів, оскільки захищене з'єднання запобігає перехопленню даних сторонніми. Не можна допускати доступу до API через незашифровані канали – треба налаштовувати сервер так, щоб автоматично перенаправляти всі HTTP-запити на HTTPS (наприклад, через 301 redirect). Це є базовою вимогою для безпечної роботи системи автентифікації.

Надійне зберігання ключів є критичним аспектом захисту JWT-токенів: у разі компрометації секретного ключа зловмисник зможе підписувати власні токени, які виглядатимуть як легітимні. Саме тому ключі мають бути доступні лише на сервері, захищеному обмеженими правами доступу, а також регулярно змінюватися у рамках ротації.

Усі секретні ключи для хешування паролів, підписання токенів та інші, а також параметри конфігурації додатку потрібно зберігати у надійному сховищі. Наприклад використовувати віртуальне оточення (VENV). Такий підхід дозволяє ізолювати налаштування середовища від основного коду додатку, запобігаючи випадковому витоку конфіденційної інформації – наприклад, через публічні репозиторії або журнали логів. Проте сам по собі VENV не є повноцінним сховищем для секретів, а радше інструментом для керування залежностями та змінними середовища. Тому рекомендується поєднувати використання

віртуального оточення з .env файлами (які не слід додавати в Git) або з зовнішніми сервісами, як-от Vault, AWS Secrets Manager чи Docker Secrets.

JWT токени мають підпис, що гарантує їхню цілісність. Для цього завжди потрібно використовувати криптографічно стійкі алгоритми, такі як HS256 (HMAC-SHA256), RS256 (RSA-SHA256) або ES256 (ECDSA). HS256 є симетричним алгоритмом і вимагає захищеного зберігання єдиного секретного ключа на стороні сервера. RS256 і ES256 – асиметричні, де приватний ключ використовується для підпису, а публічний – для перевірки, що особливо корисно в масштабованих системах. У жодному разі не можна приймати токени з alg: none, оскільки це дозволяє зловмиснику створити непідписаний токен, який система може сприйняти як валідний. Також заборонено дозволяти клієнту самостійно задавати алгоритм у заголовку токена, оскільки це відкриває можливість для атак типу «algorithm confusion». Алгоритм перевірки завжди має бути жорстко зафіксований на стороні сервера.

Access токени повинні мати короткий термін дії – зазвичай від 5 до 15 хвилин. Це обмежує можливості зловмисника у випадку викрадення токена, оскільки вікно для атаки мінімальне. Refresh токени, навпаки, призначені для довшого зберігання (від декількох годин до декількох днів) і використовуються для генерації нових access токенів без повторної авторизації. Щоб запобігти їхньому зловживанню, refresh токени слід прив’язувати до конкретних ознак сесії, таких як IP-адреса клієнта та User-Agent. Якщо під час запиту на оновлення токена ці параметри відрізняються від оригінальних, це може свідчити про викрадення, і токен має бути відкликаний.

Слід мінімізувати payload JWT, уникаючи додавання конфіденційних або персональних даних, таких як email чи ролі, якщо вони не є необхідними на стороні клієнта. Зайва інформація збільшує розмір токена і підвищує ризики витоку даних у разі компрометації. Обмеження payload лише необхідними полями сприяє кращій безпеці та продуктивності системи.

Для кожного JWT токена слід генерувати унікальний ідентифікатор – поле jti, яке дозволяє відстежувати конкретний екземпляр токена. Це особливо корисно

при реалізації механізму відкликання токенів. Під час виходу користувача або при виявленні підозрілої активності токен із відповідним jti додається до чорного списку, наприклад у Redis. Кожен запит із токеном перевіряється на наявність його jti у цьому списку. Якщо токен було відкликано – доступ блокується. Такий підхід дає можливість гнучко керувати життєвим циклом токенів і захищає від їх повторного використання.

Щоб ускладнити використання токена на іншому пристрої або в іншому середовищі, варто прив’язувати токен до конкретного клієнта. У payload токена додаються додаткові поля: ip – IP-адреса користувача на момент видачі токена, і ua_hash – хеш значення User-Agent. Під час кожного запиту ці значення порівнюються з фактичними. Якщо IP або User-Agent не збігаються, це може вказувати на спробу використання викраденого токена. У такому випадку токен відкликається, сесія завершується, а подія фіксується як інцидент безпеки.

Обов’язково слід перевіряти всі стандартні claims у токені, щоб підтвердити його легітимність. Поле iss (issuer) має відповідати довіреному емітеру токена, aud (audience) – відповідати вашому сервісу або застосунку, для якого призначений токен. Часові поля exp (expiration), iat (issued at) та nbf (not before) необхідно перевіряти, щоб гарантувати, що токен використовується у допустимому часовому проміжку. Ігнорування цих перевірок може призвести до прийняття прострочених або передчасних токенів, що створює вразливість у системі безпеки.

Не треба зберігати JWT у LocalStorage, оскільки це робить токени вразливими до XSS-атак – зловмисник, який виконає шкідливий скрипт, зможе легко отримати доступ до токена. Набагато безпечніше зберігати токени в HttpOnly та Secure Cookie, які недоступні через JavaScript і передаються лише по захищеному HTTPS-з’єднанню. Такий підхід значно знижує ризики викрадення токенів у браузерних клієнтах.

Логування підозрілої активності має включати фіксацію IP-адреси та User-Agent користувача, спроб використання недійсних або змінених токенів, а також часті помилки авторизації. Особливу увагу слід приділяти випадкам зміни середовища виконання токена (наприклад, зміна IP або User-Agent під час сесії). Це

допомагає своєчасно виявляти спроби злому або викрадення токенів і оперативно реагувати на загрози.

ПЕРЕЛІК ПОСИЛАНЬ

1 Grassi, P.A. at al. NIST SP 800-63B. Digital identity guidelines: Authentication and lifecycle management. NIST Special Publication (SP). 2020. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf (дата звернення: 10.03.2025).

2 OWASP Top 10:2021 URL: https://owasp.org/Top10/ (дата звернення: 10.03.2025).

3 RFC 7519. JSON Web Token. 2015. RFC - Proposed Standard URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 10.03.2025).

4 Sambit K.D. Ultimate Web Authentication Handbook. Orange Education Pvt Limited, 2023. 340 p. URL: https://github.com/OrangeAVA/Ultimate-Web-Authentication-Handbook (дата звернення: 11.03.2025).

5 IBM JSON Web Token (JWT). 2025. URL: https://www.ibm.com/docs/en/cics-ts/6.x?topic=cics-json-web-token-jwt (дата звернення: 11.03.2025).

6 Introduction to JSON Web Tokens. URL: https://jwt.io/introduction (дата звернення: 11.03.2025).

7 JSON Web Token Claims. URL: https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-claims (дата звернення: 14.03.2025).

8 OWASP REST Security Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (дата звернення: 14.03.2025).

9 How OpenID Connect Works. URL: https://openid.net/developers/how-connect-works/ (дата звернення: 19.03.2025).

10 Access tokens in the Microsoft identity platform. 2025. URL: https://learn.microsoft.com/en-us/entra/identity-platform/access-tokens (дата звернення: 19.03.2025).

11 OAuth 2.0 Resource Server JWT. URL: https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html (дата звернення: 21.03.2025).
12 OWASP JSON Web Token Cheat Sheet for Java. URL: https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html (дата звернення: 21.03.2025).
13 Basic vs. JWT-Based Authentication: What's the Difference? 2025. URL: https://www.descope.com/blog/post/basic-vs-jwt (дата звернення: 02.03.2025).
14 NIST Special Publication 800-63C. Digital Identity Guidelines. URL: https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-63c.pdf (дата звернення: 22.03.2025).
15 JWT storage 101: How to keep your tokens secure. 2025. URL: https://workos.com/blog/secure-jwt-storage (дата звернення: 16.03.2025).
16 Token Lifetimes and Security in OAuth 2.0: Best Practices and Emerging Trends. URL: https://bok.idpro.org/article/id/108 (дата звернення: 22.03.2025).
17 Seven Ways To Revoke JWT Tokens. 2024. URL: https://supertokens.com/blog/revoking-access-with-a-jwt-blacklist (дата звернення: 24.03.2025).
18 RFC 8725. JSON Web Token Best Current Practices. 2020. URL: https://datatracker.ietf.org/doc/html/rfc8725 (дата звернення: 01.04.2025).
19 OWASP Logging Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html (дата звернення: 01.04.2025).